HP 9000 Series 200 Computers

HEWLETT PACKARD

BASIC 3.0 Programming Techniques



Manual Organization

Chapter 1

Welcome

This manual is intended to introduce you to the Series 200 computer's BASIC programming language and to provide some helpful hints on getting the most utility from it. Although this manual assumes that you have had some previous programming experience, you need not have a high skill level, nor does your previous experience need to be in BASIC. If you have never programmed a computer before, it will probably be more comfortable for you to start with one of the many beginner's text books available from various publishing companies. However, some beginners may find that they are able to start in this manual by concentrating on the fundamentals presented in the first few chapters. If you are a programming expert or are already familiar with the BASIC language of other HP desktop computers, you may start faster by going directly to the *BASIC Language Reference* manual and checking the keywords you normally use. You can always come back to this manual when you have extra time to explore the computer's capabilities, or if you bump into an unfamiliar concept.

Durability is a built-in feature of this easy-to-operate computer, so don't be afraid to test it. After reading each section and trying the examples shown, try your own examples. Experiment. You cannot damage the computer by pressing the wrong keys. The worst that can happen is an error message will appear. These messages help you learn its language and needs by communicating with you. All error codes are listed at the back of this manual.

What's In This Manual?

No matter what your skill level, it is helpful to understand the contents and organization of this manual. First of all, there are some things that it is **not**. Because it is organized by topics and concepts, it is not a good place to find an individual keyword in a hurry. Keywords can be found using the index, but even so, they are often imbedded in discussions, contained in more than one place, or only partially explained. Also, this is not a good place to find complete syntactical details. Program statements are often presented only in the form that applies to the specific concept being discussed, even though there may be other forms of the statement that accomplish different purposes. If you want to quickly find the complete formal syntax of a keyword, use the *BASIC Language Reference*. It is specifically intended for that purpose.

This manual contains explanations and programming hints organized topically. A program performs various "sub-tasks" as it completes its overall job. Many of these tasks can, or should be, viewed separately to be understood more easily and used more effectively. For example, perhaps you have experience in another programming language. You know exactly what a "loop" does, but you didn't find the statement you were looking for in the *BASIC Language Reference*. In the chapter on "Program Flow", there is a section called "Repetition" which explains the kinds of loops available and all the statements needed to create them. The following is an overview of the chapters in this manual.

Chapter 1: Manual Organization

Chapter 2: Entering, Running, and Storing Programs

This chapter explains the mechanics of the programming process. It discusses ways to type in a program, modify it, run it, print it, make it more readable, and save it on a disc so you can continue improving it tomorrow.

Chapter 3: Program Structure and Flow

This chapter tells how the computer finds its way around your program and offers ideas on getting it to follow the proper path efficiently.

Chapter 4: Numeric Computation

This chapter covers mathematical operations and the use of numeric variables. It includes discussions on types of variables, expression evaluation, arrays, and methods of managing data memory.

Chapter 5: String Manipulation

Although string data can be used for any purpose the programmer desires, it is most often used for the processing of characters, words, and text. Since words are more pleasant than numbers to humans, skillful use of strings can make the input and output of programs much more natural to those using the programs. This chapter explains the programming tools available for processing string data.

Chapter 6: User-Defined Functions and Subprograms

An outstanding feature of this language is its ability to change program contexts and the speed with which it can do so. Alternate contexts (or environments) are available as user-defined functions or subprograms. These are discussed in this chapter.

Chapter 7: Data Storage and Retrieval

This chapter shows many of the alternatives available for storing the data that is intended as program input or created as program output. Topics range from convenient ways to define constants to a discussion of the computer's unified mass storage system.

Chapter 8: Using a Printer

This chapter tells how to connect and use an external printer. Also covered are the formatting techniques (useful on both printer and CRT) to create organized, highly-readable printouts.

Chapter 9: Using the Real-Time Clock

An accurate real-time clock is available with timing resolution to the hundredth of a second and a range of years. Its capabilities are covered in this chapter.

Chapter 10: Communicating with the Operator

It is very fustrating for operator and programmer alike when the operator cannot figure out what is expected next, or the program crashes every time a wrong key is pressed. The chapter presents some programming techniques that help ease the interaction between the computer and a human operator.

Chapter 11: Error Handling

This chapter discusses techniques for intercepting (or trapping) errors that might occur while a program is running. Many errors can be dealt with easily by a programmer. Error trapping keeps the program running and provides valuable assistance to the computer operator.

Chapter 12: Program Debugging

We all wish that every program would run perfectly the first time and every time. Unfortunately, there is little evidence in real life to support that fantasy. The next best thing is to get the computer to do most of the debugging work for you. This chapter explains the powerful debugging features available on the computer.

Chapter 13: Efficient Use of the Computer's Resources

Which takes longer, calculating a square root or raising a number to the .5 power? Does a program run faster if the variable names are shorter? If you have a time-critical or memory-critical application, you will be interested in these answers and others provided in this chapter.

Chapter 14: Graphics

Another of the computer's significant features is its fast, precise graphics capability. Graphics is very useful for generating results in a human-compatible form. This chapter introduces the techniques of programming with graphics.

Chapter 15: Translating Programs

This chapter helps the user who is transporting programs from previous versions of BASIC. It discusses changes and enhancements.

This is a manual of programming techniques, helpful hints, and explanations of capabilities. It is not a rigorous derivation of the BASIC language. Any statements appropriate to the topic being discussed are included in each chapter, whether they have been previously introduced or not. Since most users will not read this manual from cover to cover anyway, the approach chosen should not present any significant problems. In those cases when you have difficulty getting the meaning of certain items from context, consult the Index to find additional information.

BASIC 3.0 Programming Techniques

for the HP 9000 Series 200 Computers

Manual Part No. 98613-90010

© Copyright 1984, Hewlett-Packard Company.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this document is subject to change without notice.

Restricted Rights Legend

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in paragraph (b)(3)(B) of the Rights in Technical Data and Software clause in DAR 7-104.9(a).



Hewlett-Packard Company

3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

May 1984...First Edition

Warranty Statement

Hewlett-Packard products are warranted against defects in materials and workmanship. For Hewlett-Packard Fort Collins Systems Division products sold in the U.S.A. and Canada, this warranty applies for ninety (90) days from the date of delivery.* Hewlett-Packard will, at its option, repair or replace equipment which proves to be defective during the warranty period. This warranty includes labor, parts, and surface travel costs, if any. Equipment returned to Hewlett-Packard for repair must be shipped freight prepaid. Repairs necessitated by misuse of the equipment, or by hardware, software, or interfacing not provided by Hewlett-Packard are not covered by this warranty.

HP warrants that its software and firmware designated by HP for use with a CPU will execute its programming instructions when properly installed on that CPU. HP does not warrant that the operation of the CPU, software, or firmware will be uninterrupted or error free.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

* For other countries, contact your local Sales and Support Office to determine warranty terms.

Table of Contents

Chapter 1: Manual Organization

Welcome 1
What's in This Manual 2

Chapter 2: Entering, Running, and Storing Programs

Introduction 5
Some BASIC Vocabulary 5
What Is a Program? 5
What Is a Command? 6
Enter or Execute 6
Using EDIT Mode 7
Getting Into EDIT Mode 7
Editing the Current Line 8
Entering Program Lines 8
Inserting Lines 9
Deleting and Recalling Lines 10
Renumbering a Program 10
Listing a Program 11
Using Comments 11
Getting Out of EDIT Mode 14
Running a Program 15
Prerun 15
Normal Program Execution 15
Non-Executed Statements 16
Live Keyboard 16
Wholesale Program Editing 20
Moving Program Segments 20
Copying Program Segments 21
Search and Replace Operations 21
Using Subprogram Libraries 23
Indenting 23
Cross-references 26
Program Storage and Retrieval 27
Find a Usable Volume 27
Recording a Program 28
Retrieving a Program 30
Autostart of a PROG File 35
System Configuration 37
Loading BINs 37
Storing the System 37
Scratching BINs 37
Other Mass Storage Operations 38

Softkeys 39
Default Softkey Definitions 39
Defining Softkeys 39
Softkey Definitions 41
Listing Softkey Definitions 41
Softkey Files 41
Defining Softkeys Programmatically 42
Clearing the Computer 43

Chapter 3: Program Structure and Flow

Introduction 45
The Program Counter 45
Sequence 46
Linear Flow 46
Halting Program Execution 46
Simple Branching 48
Selection 51
Conditional Execution of One Segment 51
Choosing One of Two Segments 54
Choosing One of Many Segments 56
Repetition 59
Fixed Number of Iterations 59
Conditional Number of Iterations 61
Arbitrary Exit Points 63
Event-Initiated Branching 66
Enabling Events 66
Using the Knob 68
Deactivating Events 69
Disabling Events 71

Chapter 4: Numeric Computation

Introduction 73
Numeric Data Types 73
Internal Numeric Formats 75
Evaluating Scalar Expressions 77
The Hierarchy 77
Operators 79
Resident Numerical Functions 82
Array Operations 85
Dimensioning an Array 85
Finding Out the Dimensions of an Array 91
Filling an Array 92
Extracting Values From Arrays 94
Redimensioning Arrays 95
Arrays and Arithmetic Operators 96
Boolean Arrays 98
Reordering Arrays 99
Sorting Arrays 100

Matrices and Vectors 103
Solving Simultaneous Equations 108
Singular Matrices 110
The Determinant of a Matrix 111
Ill-Conditioned Matrices 112
Using Arrays for Code Conversion 116

Chapter 5: String Manipulation

Introduction 119
String Storage 120
String Arrays 120
Evaluating Expressions Containing Strings 121
Evaluation Hierarchy 121
String Concatenation 121
Relational Operations 121
Substrings 122
Single-Subscript Substrings 122
Double-Subscript Substrings 123
Special Considerations 123
String-Related Functions 125
String Length 125
Substring Position 125
String-to-Numeric Conversion 126
Numeric-to-String Conversion 127
CRT Character Set 128
String Functions 129
String Reverse 129
String Repeat 129
Trimming a String 129
Case Conversion 130
Searching and Sorting 131
MAT Functions and String Arrays 132
Reordering an Array 135
Searching for Strings 136
Number-Base Conversion 138
Introduction to Lexical Order 139
Why Lexical Order? 139
How it Works 139
The ASCII Character Set 140
The Extended Character Set 143
Predefined Lexical Order 145
Lexical Tables 146
Notation 146
ASCII Lexical Order 146
FRENCH Lexical Order 148
GERMAN Lexical Order 150
SPANISH Lexical Order 152
SWEDISH Lexical Order 154
User-defined LEXICAL ORDER 156

User-Defined Lexical Orders 158
Sequence Numbers 159
Mode Entries 159
Bits, Bytes, and Mode Types 160

Chapter 6: User-Defined Functions and Subprograms

Introduction 165
Some Startup Details 166
Location 166
Naming 166
For Example 166
The Difference Between a Function and a Subprogram 167
REAL Precision Functions and String Functions 168
Calling and Executing a Subprogram 169
Communication 170
Context Switching 176
Live Keyboard 178
Speed Considerations 178
Using Subprogram Libraries 180
Loading Subprograms One at a Time 180
Loading Several Subprograms at Once 180
Loading Subprograms Prior to Execution 180
Deleting Subprograms Programmatically 181
Editing Subprograms 182
SUBEND and FNEND 183
Recursion 184
Top-Down Design 185
The Problem 185
A Data Structure 185

Chapter 7: Data Storage and Retrieval

Storing Data in Programs 194
Storing Data in Variables 194
Data Input by the User 194
Using DATA and READ Statements 194
Mass Storage Tutorial 198
What is Mass Storage? 198
The Structure of Model 226/236 Discs 198
The Structure of Data Files 203
Mass Storage Techniques 208
Initializing a Disc 208
Disc Labels 209
Overview of Mass Storage Access 210
Media Specifiers 211
Internal Disc Drives 214
External Devices Available 214
Non-Disc Mass Storage 220
Accessing Files 222

Reading and Writing BDAT Files 225
System Sector 225
Defining Records 225
Choosing A Record Length 226
EOF Pointers 228
Writing Data 229
Serial OUTPUT 229
Random OUTPUT 234
Reading Data From BDAT Files 237
Single-Byte Access 239
General Mass Storage Operations 240
Trapping EOF and EOR Conditions 240
Protecting Files 242
Securing Program Lines 243
Copying Files and Volumes 244
Purging Files 245
Accessing Directories 246
Extended Access of Directories 247

Chapter 8: Using a Printer

Introduction 253
Fundamentals 253
Device Selectors 254
Primary Addresses 254
Using Device Selectors 256
Using the External Printer 257
Control Characters 257
Escape-Code Sequences 258
Formatted Printing 259
Using Images 260
Special Considerations 264

Chapter 9: The Real-Time Clock

Introduction 266
Clock Time 266
Setting the Time 268
Setting the Date 270
Using the Routines 272
Day of the Week 273
Days Between Two Dates 273
Interval Timing 274
Accuracy 275
Powerfail Protection 275
Branching on Clock Events 276
Cycles and Delays 276
Time of Day 277
Priority Restrictions 277
Branching Restrictions 279

Chapter 10: Communicating with the Operator

Introduction 281
Displaying and Prompting 282
Determining Screen Width 282
Clearing the CRT 282
An Expanded Softkey Menu 285
Moving a Pointer 289
Accepting Keyboard Input 292
Get Past the First Trap 292
Entering a Single Item 293
LINPUT with Multiple Fields 296
Yes and No Questions 297
Custom Interfaces 298

Chapter 11: Error Handling

Introduction 299
Anticipating Operator Errors 300
Boundry Conditions 300
REAL Numbers and Comparisons 301
Error Trappomg 302
ON/OFF ERROR 302
ERRN/ERRL/ERRM$ 302

Chapter 12: Program Debugging

Introduction 307
Using Live Keyboard 308
Stepping 310
Tracing 311
TRACE ALL 311
PRINTALL IS 313
TRACE PAUSE 313
TRACE OFF 313
The CLR I/O Key 314

Chapter 13: Efficient Use of the Computer's Resources

Introduction 315
Data Storage 315
Data Storage in Read/Write Memory 315
Data Storage on Mass Memory Devices 316
Comments and Multicharacter Identifiers 317
Variable and Array Initialization 317
Mass Memory Performance 318
Program Files 318
Data Files 318
Benchmarking Techniques 319

INTEGER Variables 321
Minimum and Maximum Values 321
Mathematical Operations 321
Loops 322
Array Indexing 323
REAL Numbers 324
Minimum and Maximum Values 324
Type Conversions 324
Constants 325
Polynomial Evaluations 325
Logical Comparisons for Equality on REAL Numbers 326
Saving Time 327
Multiply vs. Add 327
Exponentiation vs. Multiply and SQR 327
Array Fetches vs. Simple Variables 327
Concatenation vs. Substring Placement 327
Saving Memory 329

Chapter 14: Graphics

Introduction 331
Graphics Fundamentals 332
The First Step 332
Pencil and Paper 332
Graphic Display Units 336
Labeling the Image 337
Setting the Limits 338
User-Defined Units 340
Saving an Image 441
External Output Devices 342
Initialization 342
Graphics with Printers 344
Graphics Using HPGL 345
Graphics Using Files 346
Using the Color Output Interface 347
Color Graphics Techniques 347
Interactive Graphics 353
Compatible Devices 354
Setting Up a Session 354
Interactive Graphics Techniques 355
Example Graphics Programs 360

Chapter 15: Transporting Programs

Configuring BASIC 379
Statement Changes 380
KNOBY 380
CSUBs 380
PHYREC 381
Hardware 382

Appendix

Error Messages. 383
Second Byte of Non-ASCII Key Codes 394
US ASCII Character Codes 395

Subject Index 397

Chapter 2 Entering, Running, and Storing Programs

Introduction

The first thing a programmer needs to know is how to get a program into the computer. If your background is punched cards and batch jobs, you will be delighted to discover how easy program writing is on Series 200 computers. If you have experience on other HP desktops, you should find the computer's programming procedures familiar, with some improvements. Whatever your starting point, it makes sense to learn the mechanics of program writing before you become absorbed in a study of all the available program statements.

Some BASIC Vocabulary

What Is a Program?

A **main program** is a list of program lines, with an `END` statement on the last line. A **program line** contains at least a line number followed by a statement[1]. A **statement** is a keyword (sometimes optional) followed by any parameters, lists, specifiers, and secondary keywords that are allowed with that keyword and fit in the program line. The maximum length of a program line is 256 characters. When entering programs from the keyboard, the maximum length is reduced to two CRT lines. A **keyword** is a group of uppercase characters that is understood by the computer's language system to represent some predefined action.

The computer also allows subprograms to be appended to a main program. Subprograms are also lists of program lines, but they have special requirements for their first and last lines. Subprograms are a useful programming tool, but the computer is capable of running just fine without them. Therefore, most of the concepts in this manual are presented using examples in a main program context. Subprograms are covered in depth in "User-Defined Functions and Subprograms" Chapter.

1 A line number may be optionally followed by a line label. A line label is a name that is placed after the line number and is terminated by a colon. Chapter 3 tells more about labels.

What Is a Command?

This manual makes frequent reference to "statements" and "commands". As previously mentioned, a **statement** is a keyword followed by any other elements that are appropriate for that keyword. If a statement is placed after a line number and entered, it becomes a program line. If a statement is typed without a line number and executed, it is called a **command**. There are some commands that cannot be stored as program lines, such as those using DEL and SCRATCH. There are also statements that can't be executed as commands, such as those using DIM and RETURN. However, many statements are both programmable and executable, such as those using PRINT and CALL.

When a keyword is described, the most descriptive term (statement, command, or function) is used and any restrictions are noted in the text. Since these terms are not necessarily mutually exclusive, read more than just the one term to determine the legal uses of a keyword. For example, the use of the term "statement" does not imply that a certain keyword is not keyboard executable; but the phrase "cannot be executed as a command" clearly indicates that a keyword can only be used in a program line.

Enter or Execute

Entering a program line means that you type a line number followed by a valid statement and then press one of these keys: RETURN, ENTER, EXECUTE or EXEC. The line is stored in memory as part of a program. The line performs no function until you run the program.

Executing a command means that you type a statement (no line number) and press one of the keys listed above. The command is executed immediately. It is not stored in a program.

Depending on the keyboard, you have at least one of the keys listed. In most cases it does not matter which key you use. (In some Edit modes such as FIND there is a difference between ENTER and EXECUTE.)

KEYS

Throughout the BASIC manuals you will see a word, letter or symbol enclosed in a box. This indicates an actual key on the keyboard, or a function key shown on the menu on the display.

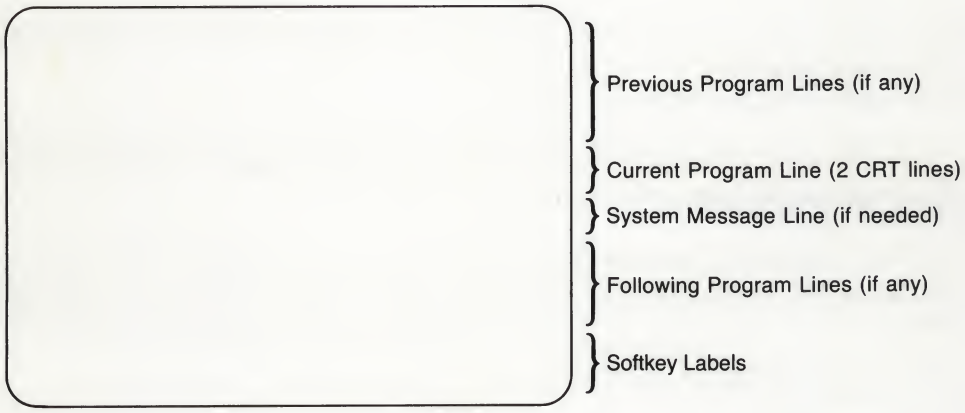
There are several different keyboards for the Series 200 computers. Each keyboard has slightly different key labels. (The BASIC User's Guide describes all keyboards in detail.) Within this manual the keys are listed once the first time they are used in a topic. Thereafter, only one of the keys is used. If you have any confusion over which key to use, refer to the BASIC User's Guide which includes a table of keys.

Using EDIT Mode

It is possible to enter a program by typing entire program lines (line number and statement) on the normal input line of the CRT and pressing the ENTER or RETURN key. There are some disadvantages to this method. First, it forces you to type in the line number (which means you have to remember what line number needs to be supplied). Second, the program lines disappear after they are entered, so it is difficult to keep track of where you are in the program and what the lines above are doing. Third, there is no way to review the program except the LIST command. Listing to a printer gives a readable copy of the program, but is slow and paper-consuming during program development. A program listed to the CRT goes by so fast that you won't be able to read anything but the end. Listing program portions to the CRT requires you to remember the line numbers of every segment you want to see. The solution to all these problems is the EDIT mode.

Getting Into EDIT Mode

To get into EDIT mode, either press the EDIT key or type the word `EDIT`, then press RETURN or EXECUTE. As a result, the format of the CRT display is transformed as shown in the following diagram.



In this mode, you can view several lines before and after the line you are editing. The system supplies the line number for the current line and program portions can be viewed by simple scrolling.

The EDIT command allows two parameters. The first is a line identifier and the second is the increment between line numbers. For example:

```
EDIT 140,20
```

This command tells the computer to place the program on the CRT so that line 140 is in the current-line position. Also, any lines that are added to the program get a line number 20 greater than the previous line.

If the increment parameter is not specified, the computer assumes a value of 10. For example:

```
EDIT 1000
```

This command tells the computer to place the program on the CRT so that line 1000 is in the current-line position, and added lines get a line number 10 greater than the previous line.

When the line identifier is not supplied, the computer has some interesting ways of assuming a line number. If this is the first EDIT after a power-up, SCRATCH, SCRATCH A, SCRATCH BIN or LOAD, the assumed line number is 10. If EDIT is performed immediately after a program has paused because of an error, the number of the line that generated the error is assumed. At any other time, EDIT assumes the number of the line that was being edited the last time you were in EDIT mode.

The line identifier also can be a line label. This makes it very easy to find a specific program segment without needing to remember its line number. For example, assume that you want to edit a sorting routine that begins with a line labeled "Go_sort". Simply execute:

```
EDIT GO_SORT
```

The line labeled "Go_sort" is placed in the current-line position and the next several lines of the routine are displayed below it.

The EDIT command is not programmable and cannot be used while a program is running.

Editing the Current Line

The BASIC User's Guide explains the keyboards and its use for typing and editing on the input line. All of these normal editing features are available in EDIT mode, plus some additional actions specific to programming. The extra EDIT mode features are explained in subsequent sections.

Entering Program Lines

Program lines are entered by typing them after the line number and pressing (ENTER), (RETURN) or (EXECUTE). Before storing the line, the computer checks for syntax errors and converts letter case to the required form for names and keywords.

Although the computer supplies a line number automatically, you are not forced to use that number if you don't want to. To change the line number, simply back up the cursor and type in the line number you want to use. This can be done to existing lines as a way of copying them to another part of the program. When you change a line number, the program is moved on the CRT so that the line just stored is one line above the current-line position. In other words, when you move a line to a new location, the new location is displayed.

Here are some points to keep in mind when changing the line numbers supplied by the computer. Changing the line number of an existing line causes a copy operation, not a move. The line still exists in its original location. Existing lines are replaced by any line entered with their same line number. Be careful that you don't accidentally replace a line because of a typing mistake in the line number.

Syntax Checking

Syntax is a term used to describe the elements that compose a line and their order. Immediate syntax checking is one big advantage of writing programs in HP BASIC instead of turning in a batch of punched cards. A great many programming errors can be detected at program entry time, which increases the chances of having a program run properly and cuts down debugging time. If the syntax of the line is proper, the line is stored, and the next line number appears in front of the cursor.

If the system detects an error in the input line, it displays an error message immediately below the line and places the cursor at the location it blames for the error. Keep in mind that there is an endless variety of human mistakes that might occur and the computer is acting on erroneous input. As a result, you might not always agree with its diagnosis of the exact error or the error's location. However, an error message is proof that something needs to be fixed. There is a complete list of error messages and their meanings in the last chapter of this manual.

Uppercase or Lowercase?

Program entry is simplified by the computer's ability to recognize the uppercase and lowercase requirements for most elements in a statement. An entire statement can be typed using all uppercase or all lowercase letters. If the statement's syntax is otherwise correct and there are no keyword conflicts, the computer places all keywords in uppercase and all variable names in lowercase with an uppercase first letter. In other words, you don't usually have to bother with the (SHIFT) key when you enter a line, because the computer knows what the line is supposed to look like. If there is a keyword conflict, an error occurs. A keyword conflict occurs when the letters of a keyword are used as an identifier (variable name, line label, or subprogram name). When this happens, simply change the case of at least one letter in the identifier name and enter the line again. A word containing a mixture of uppercase and lowercase letters is assumed to be an identifier.

The computer's assumptions about the appearance of a line won't cause any problems if your line has the proper syntax. However, if you are guessing at a keyword or syntax, don't assume that you got the line right just because the computer stored it. Take a close look at what was stored. If the computer put lowercase letters in something that you thought was a keyword, then it wasn't really a keyword. That misspelled "keyword" was perceived as a subprogram call or a variable name by the computer.

Inserting Lines

Lines can be easily inserted into a program. As an example, assume that you want to insert some lines between line 90 and line 100 in your program. Place line 100 in the current-line position and press the insert line key. The program display "opens" and a new line number appears between line 90 and line 100. Type and store the inserted lines in the normal manner. Appropriate line numbers will appear automatically. The insert mode can be cancelled by pressing the insert line key again or by performing an operation that causes a new current line to appear (such as scrolling).

While inserting lines, the computer maintains the established interval between line numbers, if possible. If the interval between lines in the preceding example was 5, the first line number appearing would be 95. When the normal interval between lines can no longer be maintained, an interval of 1 is used. Thus, after line 95 is stored, the next line number supplied is 96. When there are no line numbers available between the current line and the next line, enough of the program below the current line is renumbered to allow the insert operation to continue. In the example, this would happen after line 99 is stored. The original line 100 is renumbered to 101 and the number 100 appears in the current line.

Deleting and Recalling Lines

Lines can be deleted one at a time or in blocks. The delete line key is used to delete the current line. If delete line is pressed by mistake, the line can be recovered by pressing (RECALL), then (ENTER) or (RETURN). The computer has a recall buffer that holds the last several lines entered, deleted, or executed.

The DEL command can also be used to delete lines. When the keyword DEL is followed by a single line identifier, only a single line is deleted. The line identifier can be a line number or a line label. A combination of an EDIT command and a press of the delete line key produces the same results, but has some advantages. There is a typing aid key to start the command, you can see the line before you delete it, and the delete line key saves the line in the recall buffer (the DEL command does not). Therefore, DEL is more useful for deleting blocks of lines.

Blocks of program lines can be deleted by using two line identifiers in the DEL command. The first number or label identifies the start of the block to be deleted, and the second number or label identifies the end of the block to be deleted. The line identifiers must appear in the same order they do in the program. Here are some examples.

`DEL 100,200`	Deletes lines 100 thru 200, inclusive.
`DEL Block2,32766`	Deletes all the lines from the one labeled "Block2" to the end of the program.
`DEL 250,10`	Does nothing except generate an error.

If you have subprograms or user-defined functions in your program, they can only be deleted in certain ways. Primarily, the SUB or DEF FN statement cannot be deleted without deleting the entire subprogram or function. This is explained fully in the "User-defined Functions and Subprograms" chapter. A CSUB line can be deleted, and doing so removes the entire subprogram.

The DEL command is not programmable and cannot be used while a program is running.

Renumbering a Program

After an editing session with many deletes and inserts, the appearance of your program can be improved by renumbering. This also helps make room for long inserts. Renumbering is done by the REN command. The starting line number, the interval between lines and the range can be specified. For example:

```
REN 100,5 IN 1,500
```

This command renumbers current lines 1 thru 500, using 100 for the first line number and an increment of 5. If the increment (second parameter) is not specified, 10 is assumed. If a range is not specified, the entire program is renumbered. For example:

```
REN 1000
```

This command renumbers the entire program, using 1000 for the first line number and an increment of 10. When no parameters are specified (`REN`), 10 is assumed for the first line number and the increment, and the entire program is renumbered.

Listing a Program

All or part of your program can be displayed or printed by executing a LIST statement. The LIST statement allows parameters that specify both the range of lines to be listed and the device to which the listing should be sent. If the keyword LIST is executed without any parameters, the assumed action is to list the entire program on the system printer. The default system printer after a power-on or SCRATCH A is the CRT. (The system printer is defined by the PRINTER IS statement.)

Starting and ending line numbers can be specified in the LIST statement. The line identifiers can be labels. For example:

`LIST 100,200`	Lists lines 100 thru 200, inclusive.
`LIST 1850`	Lists the last portion of the program, from line 1850 to the end.
`LIST Rocket`	Lists the program from the line labeled "Rocket" to the end.

Directing the listing to a device other than the CRT is easy, but involves concepts that have not been introduced yet. If you want a listing on an external printer and you don't understand the brief examples here, refer to the beginning of the "Using a Printer" chapter for an explanation of device selectors. One way to get a listing on an external printer is to specify a different system printer. This is done with the PRINTER IS statement described in the "Using a Printer" chapter. However, it is often desirable to keep the CRT as system printer and still get program listings on an external printer. This is done by specifying the printer in the LIST statement. For example:

```
LIST #701
```

This statement sends the entire program listing to an HP-IB printer (address 01) without changing the system printer selection. When both the printer and the line range are specified, the printer number is specified first and terminated with a semicolon. The following example lists lines 200 thru 500 on the device serviced by interface select code 12.

```
LIST #12;200,500
```

Using Comments

When first learning how to program, most people view the use of comments, long variable names, descriptive printouts, and other documentation tools as merely extra typing that isn't really necessary in their short programs. As time passes, old programs are expanded, new programs are written, and more people use the available software. Eventually, software support activities become necessary. Some obscure bug is found or some exciting enhancement is requested. The programmer picks up a copy of a program written a year ago and can't begin to remember what "X1" was or why you would ever want to divide it by "X2". Program documentation can make the difference between a supportable tool that adapts to the needs of the users and a support nightmare that never really does exactly what the current user wants. Keep in mind that the local software support person just might be you.

The BASIC language on Series 200 computers makes it easy to write self-documenting programs. In addition to BASIC's standard REM (remark) capability, its primary documentation features are line labels, 15-character names, and end-of-line comments. Although this section deals primarily with commenting methods, all of these features work together to make a readable program. The following example shows two versions of the same program. The first version is uncommented and uses "traditional" BASIC variable names. The second version uses the features of HP's BASIC language to make the program more easily understood. Which version would you rather work with?

```
100  PRINTER IS 1
110  A=.03
120  B=.02
130  X=0
140  Y=0
150  C=A+B
160  PRINT "  Item        Total       Total"
170  PRINT "  Price        Tax        Cost"
180  PRINT "------------------------------"
190  P=0
200  INPUT "Input item price",P
210  IF P<0 THEN 290
220  D=P*C
230  E=P+D
240  X=X+D
250  Y=Y+E
260  DISP "Tax =";D;"Item cost =";E
270  PRINT P,X,Y
280  GOTO 190
290  END
```

```
100  ! This program computes the sales tax for
110  ! a list of prices. Item prices are input
120  ! individually. The tax and total cost for
130  ! each item are displayed. The running
140  ! totals for tax and cost are printed on
150  ! the CRT. Modify line 220 to change the
160  ! the system printer.
170  !
180  ! Sales tax rates are assigned on lines 230
190  ! and 240. The rates used in this version
200  ! of the program were in effect 1/1/81.
210  !
220  PRINTER IS 1                ! Use CRT for printout
230  State_tax=.03               ! Local tax rates
240  City_tax=.02
250  !
260  Total_tax=0                 ! Initialize variables
270  Total_cost=0
280  Tax_rate=State_tax+City_tax
290                              ! Print column headers
300  PRINT "  Item       Total       Total"
310  PRINT "  Price       Tax        Cost"
320  PRINT "------------------------------"
330  !
340 Get_price:                   ! Start of main loop
350  Price=0                     ! Don't change totals if no entry
360  INPUT "Input item price",Price
370  IF Price<0 THEN Done        ! Negative entry stops program
380  Tax=Price*Tax_rate
390  Item_cost=Price+Tax
400  Total_tax=Total_tax+Tax     ! Accumulate totals
410  Total_cost=Total_cost+Item_cost
420  DISP "Tax =";Tax;" Item cost =";Item_cost
430  PRINT Price,Total_tax,Total_cost
440  GOTO Get_price              ! Repeat loop for next item
450 Done: END
```

There are two methods for including comments in your programs. The use of an exclamation point is demonstrated in the second example program. The exclamation point marks the boundary between an executable statement and comment text. There does not have to be an executable statement on a line containing a comment. Therefore, the exclamation point can be used to introduce a line of comments, to add comments to a statement, or simply to create a "blank" line to separate program segments. Exclamation points may be indented as necessary to help keep the comments neat.

The REM statement can also be used for comments. The exclamation point is neater and more flexible, but the REM statement provides compatibility with other BASIC languages. The REM keyword must be the first entry after the line identifier and must be followed by at least one blank. Here are some examples of proper and improper REM statements.

RIGHT	WRONG
`10 REM Check Book Balance`	`20 REMinitialize array`
`40 Start2: REM Subtotal loop`	`50 X=PI*R^2  REM Area of circle`

Each programmer has an individual style in the use of comments. Therefore, the following is not a list of rules. It is simply some suggestions on the effective use of comments.

- Include a heading on programs that tells the purpose of the program. Why was the program written, what does it do, and who would probably be using it?
- Give any helpful support information, such as the author of the program, the revision date, where to call or write for help, and instructions for any modifications that might be made by a normal user.
- Identify all significant variables, especially global variables. A descriptive variable name may do the job, or a more detailed explanation may be needed.
- Describe any input or output devices that are required for the proper running of the program. This may even include an explanation of how to modify the program to accommodate alternate devices (when such changes are reasonable).
- Make major blocks and entry points visible. Many tools are available for this, including descriptive labels, indenting, spacing, and comments describing program flow.
- Use comments freely to describe the action of complex lines, equations, fancy manipulations, and "low-level" operations like CONTROL statements and escape code sequences. These heavily coded operations can be very important to the computer and very mysterious to the human trying to read the program.

Getting Out of EDIT Mode

There are many ways to terminate the EDIT mode. Your choice depends upon what you want to do next. If you simply want to return the CRT to its "normal" mode (input line on the bottom and printout area above), press (PAUSE), (CLR SCR) or (Clear display). Either of these keys terminates EDIT mode and returns the screen to the normal format.

Another way to leave EDIT mode is to proceed with another operation. The key choices in this case are (RESET), (RUN), (STEP), and (CONTINUE). All of these keys terminate EDIT mode and perform their normal function. A detailed list of the items affected by (RESET) is contained in the "Useful Tables" chapter at the back of the *BASIC Language Reference*. The (RUN) key starts normal program execution. This is described in the "Running a Program" section of this chapter. The (STEP) key starts single-step program execution, as described in the "Program Debugging" chapter.

CONTINUE is not always a valid way to leave EDIT mode. If you paused a program and used EDIT mode only as a means of looking at various program segments, (CONTINUE) may be pressed to resume program execution. However, if any program lines are changed while in EDIT mode, the program moves from the paused state into the stopped state. In that case, CONTINUE is not a valid operation and results in an error. This is covered in the next section, "Running a Program"

EDIT mode is also terminated by a GET or LOAD operation or by an operation that uses the CRT (for example: `CAT`).

Running a Program

The normal running of a program is started by the [RUN] key or the RUN command. Pressing the [RUN] key is equivalent to typing RUN and pressing [EXECUTE], [ENTER] or [RETURN]. This tells the computer to go through a **prerun** phase and then begin normal program execution with the lowest numbered line in the main program. The RUN command can also be followed by a line identifier that lets you specify where the program execution is to begin.

Prerun

There are three primary reasons for the prerun. The first purpose is to reserve sufficient memory for all the variables in the program (except those that are ALLOCATEd). This includes all variables in COM statements, those declared in DIM, REAL, and INTEGER statements, and all implicitly declared variables. The "Number Computation" chapter explains the declaration of numeric variables, and the "String Manipulation" chapter covers the dimensioning of string variables.

The second purpose is to locate all the context boundaries. These are defined by the END, SUB, SUBEND, DEF FN, and FNEND statements.

The third purpose of the prerun is to detect errors that involve interaction between lines. The previous section explained that the computer checks for syntax errors before it stores a program line. Although this is true, there are some errors that can't be detected by looking at a single line. For example, a program line that uses properly placed subscripts can appear to be correct when it is stored. However, if that line references three dimensions in an array that had previously been declared to have only two dimensions, it is in error. To detect an error of that kind, the computer needs to "look at" the entire program to see all the dimension statements as well as the variables used in each line. Some other examples of this kind of error are specifying a ON...GOTO to a line that does not exist or improper matching of statements like FOR...NEXT and IF...END IF.

Normal Program Execution

Program execution is not a method for destroying programs (although there are some reported cases of programmers having an urge to kill). The term **execution** is used to describe the process used by the computer while it is completing the tasks described in its program. The process of program execution is summarized below.

1. Determine which program line is to be acted on next.
2. Identify the statement that follows the line number and label (if any) on that line.
3. If the statement has a run-time action, perform the action described in the statement.
4. Repeat steps 1 thru 4 until an END, STOP, or PAUSE statement is executed.

The continuing process of determining which line is to be executed next is discussed in detail in the next chapter, "Program Structure and Flow". The RUN command determines which line is acted on first. Executing RUN with no parameters, or pressing the [RUN] key, causes the execution process to begin at the first (lowest-numbered) line of the main program. Execution can be started anywhere in the main program by using the RUN command with a line identifier. For example:

```
RUN 220
```

This command causes execution to begin at line 220, if there is such a line. If there is no line 220 in the main program, execution begins with the line whose number is closest to and greater than 220. The line identifier can also be a label. For example:

```
RUN Spot_run
```

This command causes execution to begin with the line labeled "Spot_run". If there is no such label, an error results.

Note that the prerun phase is always the same, whether the actual execution begins at the program start or somewhere in the middle. Also, if a starting line is specified, that line must be in the main program. An error 3 results if you attempt to start a program in a user-defined function or subprogram. Even if the starting point is correctly specified, be alert to the effects of starting a program in the middle. Skipping over a section of the program may result in null values for some of the variables. Although it is legal to start in the middle of a subroutine, an error is generated when the RETURN statement is executed.

Non-Executed Statements

In the preceding summary of normal execution, step 3 mentioned that only statements with run-time actions are executed. The term **run-time** refers to the state that exists after the prerun, when the computer is actually performing the actions described in the program. Some statements are not executed in the course of normal program flow, but are merely "looked at" and then bypassed. The following is a list of some statements that do not cause an action as a result of run-time execution.

- Comments and REM statements. These never cause an action.
- Variable declarations; COM, DIM, REAL, and INTEGER. These are executed during prerun and skipped over at run-time. The OPTION BASE statement is part of the declaring process.
- DATA statements. These are accessed by the READ statement, not executed.
- SUB and DEF FN statements. These are used during prerun to establish the program structure and are skipped over at run-time.
- Structuring statements, such as LOOP, END LOOP, ELSE, END IF, etc. These are matched and checked for proper nesting at prerun.

Live Keyboard

When a program is running, the keyboard is still active. Commands can be executed, variables can be inspected and changed, and the state of the computer can be changed. The term **live keyboard** is used when talking about commands that are executed during a running program. One of the principal uses for live keyboard commands is the troubleshooting and debugging of programs in the development stage. This application is covered in the "Program Debugging" chapter. The discussions presented here are intended to demonstrate the various machine states (running, paused, and stopped).

Pausing and Stopping

If the operator does not intervene, a program will run until it reaches an END, STOP, or PAUSE statement, or until it pauses to input some data or report an error. If you wish to pause or stop a program before its normal completion, the (PAUSE), (STOP), or (RESET) keys can be used. Here is a summary of the action of these keys. ((CLR I/O) has an action very similar to (PAUSE) if the computer is executing an I/O statement.)

- (RESET) — This stops the program immediately, aborting any I/O operations and resetting any interface cards. (RESET) does not affect the printout area of the CRT, program or variable memory, tabs, or the recall buffer. CONTINUE is not allowed after a RESET.
- (STOP) — This stops the program after the computer finishes executing the current line. STOP does not affect the interfaces, the CRT, program memory, the values of variables, tabs, or the recall buffer. STOP returns the program to the main context. CONTINUE is not allowed after a STOP.
- (PAUSE) — This pauses program execution after the computer finishes the current line and any I/O operations in progress. PAUSE leaves all necessary internal information intact, so that program execution can be resumed again with the (CONTINUE) key. Pressing (CONTINUE) after a PAUSE causes program execution to resume in a normal manner from the place where it was paused.
- (CLR I/O)— This aborts any I/O statement in progress and pauses the program. The program counter is returned to the beginning of the aborted I/O statement, so that CONTINUE causes the program to resume with that same statement. This is useful when the computer is "hung" trying to output to a device that is down, or for pausing a program that is executing an INPUT statement.

On an HP 46020A keyboard, STOP is (SHIFT) (STOP), PAUSE is (STOP), CLR I/O is (BREAK).

The current state of the computer is indicated in the lower right-hand corner of the CRT. The character in this corner is referred to as the "run light". The following table shows the various indications of the run light and their meaning.

Indicator	Computer State
(blank)	Program stopped; CONTINUE not allowed
▒	Program running
-	Program paused; may be continued
IO	Program paused, but a TRANSFER is still active
?	Computer is waiting for an input from the keyboard
*	Computer is executing a command from the keyboard

For Example

To demonstrate some of the interaction between a program and the keyboard, enter the following simple program.

```
10  DISP "NEXT COMMAND?"
20  X=0
30  PRINT X;
40  X=X+1
50  WAIT .1
60  GOTO 30
70  END
```

1. After you have entered the program, press (RUN) and observe the CRT. Notice that the DISP message appears in the display line, the printout area fills with a sequence of numbers, and the run light indicates that a program is running.
2. Press (PAUSE). The printout of numbers stops, and all the data on the CRT remains unchanged. The run light now indicates that the program is paused and can be continued. The program line that appears at the bottom of the CRT is the next line that will be executed when program execution resumes.
3. Press (STEP) a number of times. The program is executed one line at a time, as indicated by the lines changing at the bottom of the CRT. Notice that the program is still paused and continuable after each press of the (STEP) key. The (STEP) key can be a great help when you are trying to find certain kinds of problems. Chapter 12 "Program Debugging" gives the details of this and other debugging tools.
4. Press (CONTINUE). The printout on the CRT resumes with the next number in the sequence. The run light again indicates that a program is running.
5. Press (STOP). The printout of numbers stops, and all the data on the CRT remains unchanged. However, the run light is off, indicating a stopped condition.
6. Press (CONTINUE). An error results. A stopped program cannot be continued.
7. Press (RUN). The program runs again, but the number sequence has restarted from the beginning, not from the next number in the sequence. RUN causes the program to restart, not resume.
8. Type `X=1` and press (EXECUTE) or (RETURN). Notice that the numbers being printed start over with "1". The live keyboard was used to change the value of "X", and the program used the new value from the keyboard.
9. Press (RESET). The program stops and the data remains in the printout area, but the display line is cleared and the message `BASIC Reset` appears at the bottom of the CRT. Although the clearing of the display line seems like a minor effect, it indicates an important point. (RESET) and STOP have different effects on interfaces and peripheral devices. This aspect of (RESET) is summarized in the RESET Tables in the back of the *BASIC Language Reference* and is discussed fully in the *BASIC Interfacing Techniques* manual.
10. Press (RUN), then type `WAIT 5` and press execute. Notice that the run light changes to indicate that a keyboard command is being executed and the printout is delayed for five seconds while the live keyboard command is processed. Actually, the run light changed when the `X=1` command was executed in step 8, but it happened so fast that you didn't see it.

11. Press (PAUSE), then type `EDIT` and press (EXECUTE) or (RETURN). The display on the CRT changes to show the program. The line you were editing last appears in the current-line position. Notice that the run light is still visible in the lower right-hand corner and it indicates that the program is paused.
12. Press (CONTINUE). The CRT returns to normal mode, and the printout of numbers continues in sequence. However, the previous data on the display was lost when the CRT was used for EDIT mode.
13. Press (PAUSE), then type `EDIT 50` and press (EXECUTE) or (RETURN). The CRT changes to EDIT mode, and the program appears again. This time, line 50 is in the current-line position. Notice that the run light indicates that the program is paused. Change line 50 to `WAIT .2` and press (ENTER) or (RETURN). The new line 50 is entered, but the run light goes out. Changing the program caused it to move from the paused state to the stopped state.
14. Press (CONTINUE). An error results. As mentioned earlier, a program can be viewed while it is paused, but it cannot be changed. Once any program line has been changed, the program is no longer paused, and CONTINUE is not allowed.

This simple demonstration covers most of the highlights of live keyboard, program states, and the run light. The "waiting for input" indication can be seen when using the INPUT and LINPUT statements described in Chapter 10, "Communicating with the Operator". The "paused with TRANSFER completing" indication is not described in this manual. It is a special state that results from the use of overlapped I/O and is discussed in the *BASIC Interfacing Techniques* manual.

Wholesale Program Editing

There are some commands which make is easy to do large amounts of program editing very quickly. Among these are commands to move blocks of text, copy blocks of text, replace occurrences of one string with another string, find occurrences of a string, cross-reference the program, selectively load and delete subprograms, and more. A detailed explanation of these commands follows. Some commands require the PDEV or XREF extensions.

Moving Program Segments

Often during program development, you enter a section of code that performs some function, thinking that this function will only be needed in that one place. Sure enough, a short time later you find that you need it here and here, too. It becomes obvious that the section of code would be much better as a subprogram. But how on earth do you move those thirty-five lines of code? You certainly don't want to retype the whole thing. The non-programmable MOVELINES command is made for just this type of problem. What you need to do is:

1. Go to the end of the program (after *everything* else).
2. Enter the subprogram header (because you can't enter a SUB or DEF FN statement if there are other statements following it).
3. Move the text from its old position to a line number greater than the line number of the SUB or DEF FN statement you entered in Step 2.
4. Terminate the new subprogram with a SUBEND or FNEND.
5. Go back to the place where the code came from, and enter a line which invokes the new subprogram.

Another situation in which MOVELINES is very useful is in a large program which is developed over a period of time. You eventually come to the point where it would be nice to have the logically-connected subprograms together in their respective areas of the program. A typical example follows.

Your program has these main functions:

- Load the data
- Edit the data
- Print a report
- Plot a graph
- Store the data

The Editing, Printing, and Plotting options may each have options of their own. You'd like the Load subprogram to be followed by all the Edit subprograms, followed by all the Printing subprograms, etc. The MOVELINES command is a tremendous aid in doing this; however, there is one restriction: you cannot move a subprogram to a point where there would be lines of code after it. That is, you cannot move a subprogram to a line number which is less than or equal to the largest line number. Nevertheless, the same thing is accomplished *by moving other things to the end of the program.* Say you have subprograms A, C, D, and B in that order, and you want them in A, B, C, D order. You'd like to move subprogram B to a position between A and C. You can't do this. But you *can* do something else which amounts to the same thing: move *both* C and D to a point after B.

Note that when dealing with entire subprograms with either MOVELINES or COPYLINES (discussed next), any comments which occur after the SUBEND or FNEND must be moved/copied also.

This is a good way to insure that the many utility subprograms used by large programs can be found in an extensive listing: put them in alphabetical order.

Copying Program Segments

The non-programmable COPYLINES command is similar to the MOVELINES command, except it leaves the code in the old location also. This is desirable when you want a section of code that is very similar, but not identical to a section of code you already have. (If it were identical, you'd probably put it into a subprogram.) It is often easier to copy code and modify one version than to type two separate, only slightly different, versions. Here is an example of where COPYLINES is useful.

You are working in the Personnel department of your company, and you're writing a program which will need to be run on several Series 200 computers with standard keyboards. Both the Model 216 and the Model 236 have 80-column screens, whereas the Model 226 has a 50-column screen. This means that the softkey labels must have different lengths. On the Model 226, they are eight characters long, and on the Model 216 and Model 236, they are fourteen characters long. In the section of code you're writing you'd like the ON KEY labels to be as descriptive as possible; you want to use all fourteen characters if you have them.

```
100   IF POS(SYSTEM$("CRT ID"),"50") THEN  ! 50-columns; short labels
110     ON KEY 1 LABEL "VAC SCHD" CALL Vacation_sched
120     ON KEY 2 LABEL "SICK LV" CALL Sick_leave
130     ON KEY 3 LABEL "PAYROLL" CALL Payroll
140     ON KEY 4 LABEL "CRDT UN" CALL Credit_union
150     ON KEY 5 LABEL "WRK HIST" CALL Work_history
160   ELSE                                  ! 80-columns; long labels
170     ON KEY 1 LABEL "VACATION SCHED" CALL Vacation_sched
180     ON KEY 2 LABEL "SICK LEAVE" CALL Sick_leave
190     ON KEY 3 LABEL "PAYROLL" CALL Payroll
200     ON KEY 4 LABEL "CREDIT UNION" CALL Credit_union
210     ON KEY 5 LABEL "WORK HISTORY" CALL Work_history
220   END IF
```

As you can tell by looking at this code, lines 170 through 210 are almost identical to lines 110 through 150. Therefore, copying those five lines to the new place and editing the key labels would be much faster than retyping the entire line.

Search and Replace Operations

The non-programmable FIND command finds all the occurrences of a particular string in a program. Suppose, for example, you have a variable called "Tax" in your program, and you want to change it to either "State_tax" or "City_tax", but which one you change it to depends on the context of the statement. A FIND command finds each occurrence of the string "Tax". Once there, you can look at the statement and decide whether it should be changed to "State_tax" or "City_tax".

When a program line has been found, just edit the line in the normal way and press ENTER or RETURN. If your change results in a syntax error, correct it and press ENTER or RETURN again; the FIND is not cancelled. If you want to delete the line found, press DEL LN; the line is deleted and the FIND is immediately resumed. If you don't want to change the line, press CONTINUE, and the search resumes where it left off.

To cancel a search operation before it is finished, press ↑, ↓, or EXECUTE.

Literal replacement is done with the non-programmable CHANGE command. CHANGE is like FIND in that it looks through your program and finds occurrences of the specified string. However, it also makes a tentative change that you can confirm by pressing ENTER or RETURN, or deny by pressing CONTINUE. If you are positive that you don't need to verify each replacement, appending ;ALL will cause the search-and-replace to be done with no further user intervention.

Here is an example where you replace one variable name with *two* variable names: You discover, to your dismay, that after writing twenty-five subprograms, all of which use a particular COM statement, that you need another variable in the middle of that COM statement. The CHANGE command just cries out to be used in this case. Say, for example, that your COM statement looks like:

```
COM /Hardware/ Plotter_is,Plotter_spec$,Printer_is
```

where `Plotter_is` is the currently specified device selector for the plotter, `Plotter_spec$` is the plotter specifier and `Printer_is` is the device selector for the printer. You want to put `Dump_dev` into the COM statement, right after the variable `Plotter_spec$`. Your CHANGE command could look like this:

```
CHANGE "_spec$,Pri" TO "_spec$,Dump_dev,Pri" IN 1,32766
```

Note that the string you're changing **from** contains a comma. This helps narrow down the number of occurrences that match the string selected and it is desirable for this reason: if you just changed `Plotter_spec$` to `Plotter_spec$,Dump_dev`, the computer would change all the occurrences you *want* to be changed, but it would also change many you *don't* want to change. In this case, you would also cause all the places which either use or define `Plotter_spec$` to be changed, even though you wouldn't want them to be. You would change

```
Plotter_spec$="INTERNAL"
```

to

```
Plotter_spec$,Dump_dev="INTERNAL"
```

and this, of course, would cause a syntax error. You will learn efficient ways to specify searches after using the command several times.

To search the entire program, you could either go to the top of program memory with a SHIFT-↑ or use the line range of `IN 1,32766`. The former method is faster, and fewer keystrokes, but if you need to stay in a particular place in memory in order to type a long search key, the latter method is useful.

Using Subprogram Libraries

Often, a programmer has a program which is quite large, along with sizable data arrays, and an `Error 2: Memory Overflow` becomes all too common an occurrence. And neither the program nor the data can be reduced in any way. There are two keywords, `LOADSUB` and `DELSUB` which address this problem. They are mentioned here because they *are* part of Entering, Running and Storing Programs, which is the subject of this chapter, however, they are not necessary to get started. This subject will be covered later.

See Chapter 6 "User-defined Functions and Subprograms" for a detailed discussion on subprogram libraries.

Indenting

INDENT is a non-programmable command which scans the entire program and indents in appropriate places. What is meant by "appropriate places" is this: Whenever there is the beginning or end of a program statement which causes looping, is conditionally executed, or is a separate program segment (subprogram), the first character of each program line *contained* in that segment—**excluding** the line number—is moved to the right or left to make the structure of the program more intuitively obvious. For a list of how each kind of statement affects the indentation, see the *BASIC Language Reference* manual.

An example dealing with the INDENT command's capabilities follows. This program is not to be noted for its efficiency (or lack thereof—the conditions could better be checked with a SELECT statement); it is merely for the purpose of demonstrating the INDENT command. The program is shown after having been indented with various parameters. Notice how the indented structures make it easier to understand the logic flow.

This example was indented with the command

```
INDENT 7,2
```

```
10    FOR I=1 TO 5
20      REPEAT
30        INPUT "How old are you?",Age
40        Reasonable=1  ! Assume they're telling the truth...
50        IF Age<0 THEN
60          DISP "Aw, c'mon!  You can't be ";Age;"years old.  You gotta be born!
"
70          Reasonable=0
80        ELSE
90          IF Age>120 THEN
100           DISP "Oh, pshaw!  I don't believe you."
110           Reasonable=0
120         ELSE
130           IF Age>100 THEN
140             DISP "Hmm...you're well nigh a fossil, huh?"
150           ELSE
160             IF Age>60 THEN
170               DISP "Wow!  Most people your age don't use computers much."
180             ELSE
190               DISP "Glad to meet you."
200             END IF
210           END IF
220         END IF
230       END IF
240       WAIT 4
250     UNTIL Reasonable
260     DISP "You were";Age*365.242198781;"days old on your last birthday."
270     WAIT 3
280   NEXT I
290   END
```

And this example was indented with the command

```
INDENT 10,1
```

```
10        FOR I=1 TO 5
20         REPEAT
30          INPUT "How old are you?",Age
40          Reasonable=1 ! Assume they're telling the truth...
50          IF Age<0 THEN
60           DISP "Aw, c'mon!  You can't be ";Age;"years old.  You gotta be born!
"
70           Reasonable=0
80          ELSE
90           IF Age>120 THEN
100           DISP "Oh, pshaw!  I don't believe you."
110           Reasonable=0
120          ELSE
130           IF Age>100 THEN
140            DISP "Hmm...you're well nigh a fossil, huh?"
150           ELSE
160            IF Age>60 THEN
170             DISP "Wow!  Most people your age don't use computers much."
180            ELSE
190             DISP "Glad to meet you."
200            END IF
210           END IF
220          END IF
230         END IF
240         WAIT 4
250        UNTIL Reasonable
260        DISP "You were";Age*365.242198781;"days old on your last birthday."
270        WAIT 3
280       NEXT I
290       END
```

If one generalizes from the previous examples, the question arises: "What happens if the indented code goes completely off the right edge of the screen?" There are two ways that this could happen: either the starting column parameter is too large, or the number of nesting levels is too great for the specified indentation increment. If, for either of these two reasons, a line of code gets longer than the listable length of the machine, an asterisk (*) is placed immediately after the line number of that line, and the right end of the line is not visible.

To correct a problem of program lines longer than the listable length of the machine, execute another INDENT statement whose parameters are smaller.

A program which contains lines longer than 256 characters still executes, STOREs and LOADs properly, but if you SAVE the program without correcting the problem, it will not syntax properly when you do a GET operation. What is sent to the file is the line number, the asterisk, and as much of the program line as possible. The asterisk will prevent the proper syntaxing of the statement when the GET is attempted, in addition to the fact that part of the statement is missing.

Note that you can create a program on one model computer using the maximum line length and then get that program and run it on a model with a smaller CRT. You cannot, however, modify any program line which is longer than the maximum length for the current CRT without shortening it.

When indentation parameters attempt to force program statements to start too far to the right, they are bounded by the width of the screen minus eight characters. That is, the first character of a program line (excluding the line number) will never start to the right of the screen width – 8. When this is attempted, there may be several lines of code (which *should* differ in indentation) starting in column screen width – 8. Therefore, until the nesting level gets back down to a manageable point, indentation will be disabled. Note, however, that an internal indentation counter is maintained, so statements at the same nesting level will continue to have matching indentation. See the example below which was indented with the command

```
INDENT 10,15
```

Note that this was done on a machine with an eighty-column screen. Had it been done on a Model 226, with its fifty-column screen, the right-hand limit would have been reached more quickly.

```
10        FOR I=1 TO 5
20                        REPEAT
30                                    INPUT "How old are you?",Age
40                                    Reasonable=1! Assume they're telling the
truth...
50                                    IF Age<0 THEN
60                                                  DISP "Aw, c'mon!  You can'
t be "&Age&"years old.  You gotta be born!"
70                                                  Reasonable=0
80                                    ELSE
90                                                  IF Age>120 THEN
100                                                               DISP "Oh, p
shaw!  I don't believe you."
110                                                               Reasonable=
0
120                                                 ELSE
130                                                               IF Age>100
THEN
140                                                                 DISP "Hmm
...you're well nigh a fossil, huh?"
150                                                               ELSE
160                                                                 IF Age>60
 THEN
170                                                                 DISP "Wow
!  Most people your age don't use computers much."
180                                                                 ELSE
190                                                                 DISP "Gla
d to meet you."
200                                                                 END IF
210                                                               END IF
220                                                 END IF
230                                   END IF
240                                   WAIT 4
250                       UNTIL Reasonable
260                       DISP "You were"&Age*365.242198781&"days old on your last
 birthday."
270                       WAIT 3
280       NEXT I
290       END
```

Observe that there are several lines (140 and 160 through 200) which should be indented farther to the right. But since the column position of the screen width minus eight is the boundary on the right edge, they are not permitted to go as far right as they "should." Note, however, that indentation recovers and is still correct after the point at which they attempt to exceed the right-hand limit.

Retrieving a Program

Programs saved in an ASCII file are retrieved with the GET statement. Programs stored in a PROG file are retrieved with the LOAD statement. These statements can be executed from the keyboard as commands or included in a program. When executed as commands, they are used to bring a program into the computer's memory so that it can be edited or run. When included in a program, they are used to link together the segments of large programs.

To retrieve a program you need to know the name and type of the file in which it is stored. If you are not sure of either of these, execute a CAT command. The catalog display shows the name and type of all files on the disc. The options available for ASCII files are discussed first.

Using GET as a Command

The GET command is used to bring in programs or program segments from an ASCII file, with the options of appending them to an existing program and/or beginning program execution at a specified line. To clear any existing program from the computer's memory and bring in the contents of an ASCII file, the command is simply the keyword GET followed by the file name. For example:

```
GET "FORMULA"
```

This command clears the BASIC program memory and brings in the contents of the ASCII file called "FORMULA", assuming that the file contains valid program lines. If the first line does not start with a valid line number, the GET is not performed and an error 68 is reported. If the file is not an ASCII file, the GET is not performed and an error 58 is reported.

Assuming that the file contains valid program lines that were placed in the file by a SAVE operation, and their line numbers are still valid after any renumbering that is specified, the lines will be entered into program memory. If there is a syntax error in any of the program lines in the file, the lines in error are turned into comments, an error 68 is reported, and the syntax error message is sent to the system printer. This might happen if the program was written and saved on a computer that had a different version of BASIC than the one being used for the GET operation.

To append the contents of an ASCII file to an existing program, a line identifier is added to the GET command. For example, assume that there is a program already in the computer that ends with line number 740, and you want to append the contents of a file called "George". The following command could be used.

```
GET "George",750
```

This appends the program lines from file "George" to the existing program, renumbering them to start with line number 750. If the command `GET "George",100` were used in the same situation, all existing program lines from 100 to the end of the program in memory would be deleted and the contents of file "George" would be appended to the lines that remained at the beginning of the program. The program lines in file "George" would be renumbered to start with line 100.

Sometimes it is not possible to enter a line into the program. This can happen, for example, if the specified renumbering would create an invalid line number. In these cases, the line in error is sent to the system printer with an error message, but it is not entered into program memory.

The GET command can also specify that program execution is to begin. This is done by adding two line identifiers: one specifies the placement and renumbering just described, and the other specifies the line at which execution is to begin. For example, assume that there is no program in memory and that an ASCII file "RATES" contains valid program lines. A typical command to bring the contents of this file into memory and begin execution at the first line is:

```
GET "RATES",10,10
```

If there is already a program in memory, an append and run is allowed. For example:

```
GET "RATES",250,100
```

This command specifies that any existing lines from 250 to the end are to be deleted, the contents of file "RATES" is to be renumbered and appended beginning at line 250, and then normal program execution is to begin at line 100. Although any combination of line identifiers is allowed, the line specified as the start of execution must be in the main program segment (not in a SUB or user-defined function). Execution will not begin if there was an error during the GET operation.

Using GET in a Program Line

The GET statement can be used in a program to transfer execution from one program segment to another. When used in a program line, the actions of the GET statement are the same as those described for the GET command, except as noted in the following paragraphs. Two examples of a programmed GET are shown here. One demonstrates a simple linkage of two program segments, as might occur when the entire program is too long to fit in memory. The second shows a simple example of keeping a file manager in memory to GET and run various routines.

A large program can be divided into smaller segments that are connected by using a GET statement to move from one to the next. The following short example shows this technique.

First Program Segment:

```
10  COM Ohms,Amps,Volts
20  Ohms=120
30  Volts=240
40  Amps=Volts/Ohms
50  GET "wattage"
60  END
```

File "wattage":

```
10  COM Ohms,Amps,Volts
20  Watts=Amps*Volts
30  PRINT "Resistor Ohms =";Ohms
40  PRINT "Resistor Wattage =";Watts
50  END
```

One important point to note is the use of a COM statement. The COM statement places variables in a section of memory that is preserved during the GET operation. Since the program saved in the file "wattage" also has a COM statement that contains three REAL scaler variables, the values assigned to those variables in the previous program segment are preserved. If the program segments did not contain equivalent COM statements, all variables in the mismatched COM blocks would be rendered undefined by the prerun that is preformed after the GET operation. Therefore, to effectively use the GET statement to link program segments, all variables that need to be preserved must be placed in COM statements, and all program segments using those variables must have equivalent COM statements.

Note also the form of the GET statement. If this particular statement were executed from the keyboard, no program execution would take place. However, the computer understands that a GET in a program is meant to cause execution to resume. When no parameters are specified in a programmed GET, the entire old program segment is replaced by the new segment, and execution resumes with the first line in the new program segment.

If a single line identifier is used in a programmed GET (such as `GET "format",150`), the last part of the old program segment is deleted (from the specified line to the end), the new program segment is renumbered and appended to the remaining portion of the old segment, and execution resumes with the first line of the resulting program. If two line identifiers are specified, the action is the same as described for the GET command from the keyboard. In these cases, it is usually not necessary to repeat the COM statement in the second program segment. Since the COM statement is usually in the first (undeleted) part of the program, it remains after the second segment is in place. If there are two identical COM statements in the same program context, an error 12 results. Program segments that are linked in with the GET statement should repeat the original COM statement only if that original COM statement is deleted as a result of the GET operation.

An example of programming a GET statement with two line identifiers is a "file executive" program that gets and runs other program files. This technique is sometimes used when most of the computer's memory is needed to store data and the various program files perform individual operations on that data. A small example of the general technique follows. Admittedly, the tasks shown are all simple enough to be contained in a single program with plenty of room left for data. The example merely demonstrates the structures involved with this type of file linking.

In this example, a large portion of memory is used to hold weights input from an electronic scale. The "file executive" gives the operator a menu of operations to choose from and gets the requested file. The requested file performs its task and returns control to the "executive". The individual tasks shown here are the printout of the weights and the statistical analysis of the weights. Assumed, but not shown, is a routine that entered the weights from the scale and stored them in the array. (The input of data from external devices is covered in the *BASIC Interfacing Techniques* manual.)

Main Executive:

```
100   ! This program manages the utility files for
110   ! some hypothetical data. Data was stored
120   ! in the REAL array "Weights". The device
130   ! selector for the external printer is
140   ! is assigned in line 200. Each utility
150   ! file sends control back to line "Start"
160   ! when it is done. The program disc must be
170   ! present when this executive is used.
180   !
185   OPTION BASE 1
190   COM Weights(5000),Samples,Printer
200   Printer=701              ! External printer for data
210   Samples=5000             ! Also see array declaration
220   !
230 Start:                     ! Main entry point
240   PRINTER IS 1             ! For program messages
250   PRINT
260   PRINT "Enter P to print weights"
270   PRINT "Enter A for an analysis"
280   !
290 Ask:  INPUT "Enter Command Letter",In$
300   IF In$="P" THEN GET "Printout",330,330
310   IF In$="A" THEN GET "Analysis",330,330
320   GOTO Ask                 ! Incorrect entry
330   END
```

File "Printout":

```
100   ! This routine prints the data in the array
110   ! "Weights" to an external printer.
120   ! Necessary variables are initialized in
130   ! the main executive.
140   !
150   PRINTER IS Printer       ! Use external printer
160   FOR I=1 TO Samples       ! Print all weights
170      PRINT "Sample #";I;" weighs";Weights(I)
180   NEXT I
190   PRINT CHR$(12)           ! Form-feed
200   GOTO Start               ! Return to executive
210   END
```

File "Analysis":

```
100   ! This routine finds the mean and standard
110   ! deviation of the data in "Weights".
120   ! Necessary variables are initialized in
130   ! the main executive.
140   !
150   Sumx=0                    ! Clear sum of X
160   Sumx2=0                   ! Clear sum of X squared
170   FOR I=1 TO Samples        ! Calculate summations
180      Sumx=Sumx+Weights(I)
190      Sumx2=Sumx2+Weights(I)^2
200   NEXT I
210   Mean=Sumx/Samples
220   Stddev=SQR((Sumx2-Sumx^2/Samples)/(Samples-1))
230   PRINT
240   PRINT "Number of samples =";Samples
250   PRINT "Mean weight =";Mean
260   PRINT "Standard deviation =";Stddev
270   GOTO Start                ! Return to executive
280   END
```

Notice that any information that is shared by all the routines is placed in COM. The individual task files do not contain COM statements because the COM statement in the executive routine is never deleted. Values that are shared by all routines are initialized by the executive. In that manner, a standard characteristic can be changed in the executive, with no alterations required in any of the other files. The "shared" values used in this example are the number of weights in the array (Samples) and the device selector of the external printer (Printer).

Using LOAD as a Command

The LOAD command is used to bring in programs from a PROG file, with the option of beginning program execution at a specified line. To clear any existing program from the computer's memory and load the contents of a PROG file, the command is simply the keyword LOAD followed by the file name. For example:

```
LOAD "Cannon"
```

This command clears the program memory and brings in the contents of the PROG file called "Cannon". If the file is not a PROG file, the LOAD is not performed and an error 58 is reported. If any lines require a language extension that is not currently installed, those lines cannot be executed. However, the LOAD proceeds without error.

The LOAD command can also specify that program execution is to begin. This is done by adding a line identifier. For example:

```
LOAD "STONE",10
```

This command causes the computer to load the program in file "STONE" and begin execution at line 10. The line identifier may be a label or a line number, but it must identify a line in the main program segment (not in a SUB or user-defined function).

The LOAD command cannot be used to bring in arbitrary program segments or append to a main program like GET can. Subprogram segments can be appended using the LOADSUB command. This is described in Chapter 7.

Using LOAD in a Program Line
When used in a program line, the actions of the LOAD statement are the same as those described for the LOAD command, except program execution resumes whether a line identifier is specified or not. For example:

```
120  LOAD "PART2"
```

When this program statement is executed, the existing program is replaced by the contents of the PROG file called "PART2" and program execution resumes with the first line in the new program. When a line identifier is included in a programmed LOAD statement, execution resumes with the specified line after the file is loaded.

Remember to include in a COM statement any variables that must be shared by more than one program segment. The COM statement must be present before and after the LOAD if you want all the data in COM to be preserved. The section describing GET has an example of using COM to preserve data.

Autostart of a PROG File

Your computer can be configured to LOAD and RUN a program automatically when the power is switched on. To use this feature, STORE a PROG file called "AUTOST". Then when the power is switched on, the computer automatically loads that file and begins executing the program from the first line. No action is taken if there is no "AUTOST" file present.

If you want to give your program file a more meaningful name than "AUTOST", create an "AUTOST" file that simply loads the desired file. For example, if you want to autostart a program called "ALARM", store this in the "AUTOST" file:

```
10  ! This is the AUTOST file for program ALARM
20  LOAD "ALARM"
30  END
```

On a dual-drive machine like the Model 236, the autostarted program is not restricted to the space remaining on the language system disc because there are two disc drives. A short AUTOST file on drive 0 can load a long program file from drive 1. The following example shows this technique.

```
10  ! This is the AUTOST file for program ALARM
20  MASS STORAGE IS ":INTERNAL,4,1"
30  LOAD "ALARM"
40  END
```

By using the appropriate mass storage unit specifier, this general technique can be extended to use external disc drives.

The AUTOST program can also load the BIN files automatically. For example,

```
10   !This is the AUTOST file.
20   LOAD BIN "DISC:HP,702"
30   LOAD BIN "HPIB:HP,702"
40   MASS STORAGE IS ":HP,702"  !Specify if different from where SYSTM
50   LOAD BIN "MS"                    !was loaded.
60   LOAD BIN "IO"
70   LOAD BIN "GRAPH"
80   END
```

Be sure to load the drivers for the mass storage device and the card driver before you load the other BINs.

If you load BASIC from an SRM, the system looks for a file in the SYSTEMS directory named AUTOSTNN where NN is your node number. If this file is not found, the system looks for a file named AUTOST in the root directory.

System Configuration

The BASIC Language System is contained in several files. The SYSTM file contains what is sometimes called "core" BASIC. Extensions to core BASIC and device drivers are in BIN files. The language extensions add statements to BASIC. For example, the PDEV extension gives you the MOVELINES and COPYLINES commands. The device drivers allow you to use mass storage devices other than the internal discs and memory. Whenever you need an extension for an example in this manual, the BIN file name is given.

Loading BINs

As shown in the previous example, you load an extension or driver with the LOAD BIN statement.

```
LOAD BIN "GRAPHX:HP,700"
LOAD BIN "ERR"
LOAD BIN "DISC:INTERNAL,4,1"
```

If you attempt to execute a statement that requires a language extension, error 1 occurs. You can load the required BIN and then continue your programming.

You can LOAD a STOREd program that requires language extensions not present in the computer, but you cannot run it. Error 1 occurs and the option number, or extension name if ERR is loaded, is displayed. The option numbers are listed in the Useful Tables section of the BASIC Language Reference manual.

You can GET a SAVEd program that requires language extensions, but errors occur during the GET. Each statement that requires a missing option is made a comment. An ! is placed before the line. You can load the missing BIN and then edit the program.

Storing the System

Once you have all the BINs in memory, you may want to save that configuration for the next time you boot up the system. The STORE SYSTEM command stores core BASIC and all the BINs into one file.

```
STORE SYSTEM "SYSTEM_B:HP,702,1"
STORE SYSTEM "SYSTEM_2:REMOTE"
STORE SYSTEM "SYS_MINE"
```

The boot ROM looks for a SYSTM file which begins with "SYSTEM_". A 3.0 and 4.0 ROM also looks for SYSTM files which begin with "SYS_". A SYSTM file which has a different prefex cannot be booted. If there is more than one loadable file, and you press the space bar on the keyboard after power up and before the file is loaded, all loadable files are listed. You can choose the one you want loaded. If you do not press the space bar, the boot ROM loads the first SYSTM file it finds.

Since core BASIC uses most of a 5 ¼ - inch flexible disc, you cannot store more than one system on the disc. You also cannot store several BINs with core BASIC on this disc.

Scratching BINs

You can delete the BINs from memory with the SCRATCH BIN statement. This statement deletes all the BINs, except the one which drives the CRT. Note that it also scratches any program in memory.

Other Mass Storage Operations

The general mass storage capabilities of the computer are discussed in Chapter 7. Because many of the operations are applicable to program files as well as data files, certain operations are summarized here. If these brief examples do not provide sufficient information, refer to Chapter 7 for more details.

The name of a file can be changed without disturbing the file's contents. This is done with the RENAME statement. For example, to change the name of a file from "George" to "Frank", use the statement:

```
RENAME "George" TO "Frank"
```

A file entry can be removed from the disc directory with the PURGE statement. This prevents any further access to the file. For example:

```
PURGE "Myfile"
```

This statement eliminates the file "Myfile" from the disc directory.

A file can be given a protect code by using the PROTECT statement. A protect code is a 2-character string that must be specified to modify the file, but it does not appear in the catalog display. For example, to protect the file "SECRET" with the protect code "BS", use this statement:

```
PROTECT "SECRET","BS"
```

The protect code is placed after the file name to allow access. For example, to PURGE the previously protected file "SECRET", the statement is:

```
PURGE "SECRET<BS>"
```

Files can be copied with the COPY statement. Some examples:

```
COPY "MYPROG" TO "MYPROG:INTERNAL,4,1"
COPY "MYPROG:HP8290X,700,0" TO "MYPROG:HP8290X,700,1"
```

These statements create a backup copy of the file MYPROG. The first example copies from the right-hand drive to the left-hand drive on a Model 236. The second example copies a file from the left-hand drive to the right-hand drive on an external device such as the HP 82901 or HP 9121.

Softkeys

There is a set of keys on your keyboard labeled either k0 thru k9 or f1 thru f8. These are softkeys. You can define the function of these keys as typing aids. That is, you define one softkey to contain any combination of keystrokes. You may include all the ASCII characters, as well as non-ASCII keys, such as arrow keys, insert and delete character, continue, etc. This has nothing to do with programmatic ON KEY definitions; they are discussed in future chapters.

The KBD extension is required to use typing aids. When you load KBD, the typing aids are given default definitions. These default definitions include some commonly used commands. For example, k5 and f3 are defined as `SCRATCH`. k7 and f5 are defined as `CAT`. You can see the definitions in the softkey menu on your display.

If you have an HP 46020A keyboard, your softkeys are labeled f1 thru f8. There are System defined softkeys and User softkeys. You cannot change the definitions for the System keys. These keys represent physical keys on other keyboards. There are three sets of Users softkeys which gives you 24 definable softkeys. You can change any of them. In general, the menu which is displayed shows the definitions of the softkeys. To change the menu and the definition, press the SHIFT and Menu keys. To go from the User menu to the System menu (or System to User) press the User /System key.

If you have the HP 98203A keyboard (the standard keyboard for the Model 216), you have five softkeys. Each softkey can have two definitions. When you press SHIFT and a softkey, you access k0 thru k4. When you press just the softkey, you access k5 thru k9. All ten definitions are displayed on the menu.

If you have the HP 98203B keyboard (the standard keyboard for the Model 236), you have ten softkeys labeled k0 thru k9. These keys also have a shifted version. When you press SHIFT and a softkey, you access k10 thru k19. The softkey menu displays the first 10 softkey definitions. The second 10 definitions cannot be displayed.

Default Softkey Definitions

Several typing aid definitions are loaded when you load KBD. These definitions are loaded for your convienence. You can change the definitions, load your own definitions or delete the definitions. The following sections explain how you do this.

Defining Softkeys

To define a typing aid, type `EDIT KEY` and the number of the softkey you want to define. For example,

```
EDIT KEY 2
```

You could also press EDIT k2 or EDIT f2, which is the same as the EDIT KEY command.

When you execute this command, the following message is displayed:

```
Editing key 2
```

If softkey 2 is currently defined, its definition is displayed. If it is not defined, the input line is blank.

To enter the new definition, use the keyboard. Type ASCII characters directly. To get non-ASCII keystrokes, use the CTRL key and the function key you want to use. For example type the following:

CTRL-CLR LN L I S T CTRL-EXECUTE

Note

This technique will not work for some non-ASCII keys on an HP 98203A keyboard. To enter non-ASCII keystrokes, refer to the Second Byte of Non-ASCII Key Sequences table in the Useful Table section of the BASIC Language Reference manual. Press ANY CHAR 2 5 5 followed by the character associated with the desired key.

(You may have keys other than CLR LN or EXECUTE. Use the key which means clear line and execute or return.)

By pressing CTRL and CLR LN together, you get K#. CTRL-EXECUTE gives you KX. The K means you have pressed a system key. The # or X means you have pressed CLR LN or EXECUTE.

Now, press ENTER or RETURN. The definition is stored and the softkey menu changes.

Press k2 or f2.

The LIST command is entered on the input line and then executed. You may not be able to see the command displayed because it is executed immediately.

If you do not want the command executed, leave out CTRL-EXECUTE.

Besides commands, you can define the softkeys to display any sequence of characters. For example, if you are writing a program and want to separate each subprogram with a line of asterisks you could define a softkey to do this.

```
! ********************************* ENTER
```

If a CLR LN is the first character in the definition, it will not show in the label. If you want a softkey label which "looks nice" - it doesn't have a lot of inverse-video Ks in it - you can have the visible character be an aesthetically pleasing label, the next thing a CLR LN, which erases the label which prints when the key is invoked, and the next n characters be the functional "core" of what the key is supposed to do. The result of this strategy is that when you press a softkey, the aesthetically-pleasing label is displayed, immediately erased, and the characters following the label are displayed and, if you specified, processed with an ENTER, EXECUTE or RETURN, etc.

Softkey Definitions

There is 1024 bytes of memory set aside for softkey definitions. Since there is a certain amount of overhead necessary for each one, there are approximately one thousand characters available for softkey definitions.

The maximum number of characters which may be entered in a definition is two full CRT lines. If you define the softkeys on one Series 200 and store the definitions, you can load the definitions on another Series 200 even if the definitions are greater than two of the loading computer's CRT lines. You will not be able to modify the long definitions, but you can delete the old and enter a new definition. Attempting to use an overlong typing aid will cause the typing aid buffer to overflow. Some characters will be lost.

Listing Softkey Definitions

You can list the softkey definitions. All of the currently defined softkeys are listed. To list to the system printer execute:

```
LIST KEY
```

To specify a device that is not the current system printer, include a device selector in the command. To send the listing to the printer execute:

```
LIST KEY #PRT
```

The listing does not look like what you typed for the definition if you included system keys. Since most printers cannot print an inverse video K, the term System key: is listed. Each system key is listed on a separate line. For example, if you entered the definition for [k2] in the first example, your listing is:

```
Key 2:
System Key: #
LIST
System Key: X
```

Softkey Files

When you have the typing-aid softkey definitions as you want them, you can store them on a file which can be loaded at your convenience. To store the currently-defined typing-aid definitions onto a file, use a STORE KEY command. For example, the following commands store the current key definitions onto a file called AIDS:

```
STORE KEY "AIDS"
```

or

```
RE-STORE KEY "AIDS"
```

Later, when you want to load those definitions you stored, type:

```
LOAD KEY "AIDS"
```

Executing a LOAD KEY command without a file name causes the default, power-up definitions to be restored.

Defining Softkey Files Programmatically

When you type `STORE KEY "KEYS"`, the computer takes the current softkey definitions, and writes them to a BDAT file in a format which can be understood by a LOAD KEY command. The LOAD KEY command, however, doesn't know (or particularly care) *how* that BDAT file got there, as long as it's in the correct format.

The correct format for key definitions on file is this:

- The file must be a BDAT (Binary DATa) file.
- The file must be created with FORMAT OFF. This causes the data to be written to the file in internal format, not ASCII representation.
- Each key's data consists of an integer (the key number—zero through twenty-three) followed by a string—the key's value.

See the example program below. Note that all of the keys do not have to be put onto the file, nor do the key definitions sent to the file have to be in numerical order.

```
10    ! File "DoKeyFile".                    ! Name of file which holds program
20    DIM Key_value$[160]                    ! In case you have a LONG key def.
30    INTEGER Key_number                     ! Must be 16-bit key number
40    CREATE BDAT "SOFTKEYS",3               ! Create a 3-record BDAT file
50    ASSIGN @Keys TO "SOFTKEYS"             ! Open file (FORMAT OFF is default)
60    FOR I=0 TO 9                           ! First ten keys...
70      READ Key_number,Key_value$           ! Get key number and definition
80      OUTPUT @Keys;Key_number,Key_value$   ! Write them to the file
90    NEXT I                                 ! et cetera
100   ASSIGN @Keys TO *                      ! Write EOF, close the file
110   LOAD KEY "SOFTKEYS"                    ! See if it worked
120   ! --- Key data ----------------------------------------------------------
130   DATA 9,"work!",5,"that",8,"would",0,"You",4,"you",7,"thing",2,"I",1,"see?"
,3,"told",6,"this"
140   END
```

In this way, a program can define made-to-order typing-aid key definitions.

Clearing the Computer

When power is first switched on and the language system is loaded, the memory is clear and various system elements are assigned default values. (For example, the CRT is assigned as the system printer.) This condition is called the "power-on state" of the computer. A detailed list of the conditions established at power-on is given in the "Useful Tables" chapter at the back of the *BASIC Language Reference*. Turning power off, then back on again is one way to clear the computer's memory. However, this is not necessary and often is not convenient.

The most useful method of clearing the computer is to use the SCRATCH command. There are four forms of this command to allow a choice of clearing actions. The following paragraphs give the details of the choices.

`SCRATCH` — This command clears all program statements from the computer's memory. It also clears any data which has not been placed in COM (see Chapter 6 for a description of COM).

`SCRATCH C` — This command clears **all** variables from the computer's memory, including COM variables.

`SCRATCH A` — This command clears almost everything from the computer's memory. The only exceptions are the recall buffer, the real-time clock and BINs.

`SCRATCH BIN` — This command clears all BINs except the one which drives the CRT from the computer's memory. It also performs a SCRATCH A.

`SCRATCH KEY` — This allows you to clear individual softkeys, or all of them at once. If you want to scratch a particular key, you can do this two ways. The following example clears the typing-aid definition for softkey 4.

- `SCRATCH KEY 4`, or
- `SCRATCH` [k4]

To erase *all* softkey definitions, execute

```
SCRATCH KEY
```

No SCRATCH commands are allowed in a program, and they may not be executed while a program is running.

Notes

Program Structure and Flow

Chapter 3

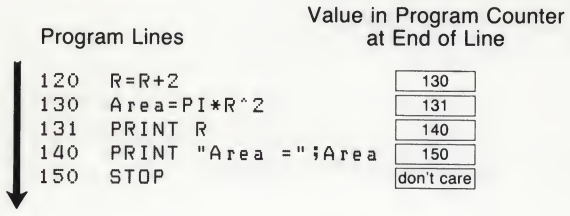
Introduction

Two of the most significant characteristics of a computer are its ability to perform computations and its ability to make decisions. If the execution sequence could never be changed within a program, the computer could do little more than plug numbers into a formula. Series 200 computers have powerful computational features, but the heart of a computer's intelligence is its ability to make decisions.

The computational power of your computer is exercised as it evaluates the expressions contained in the program lines. Chapters 4 and 5 present the various data manipulation tools available. The decision-making power is used to determine the order in which lines will be executed. This chapter discusses the ways of controlling the "flow" of program execution.

The Program Counter

The key to the concept of decision making in a computer is an understanding of the program counter. The **program counter** is the part of the computer's internal system that tells it which line to execute. Unless otherwise specified, the program counter automatically updates at the end of each line so that it points to the next program line. This is illustrated in the following drawing.



This fundamental type of program flow is called "linear flow". As shown by the arrow, you can visualize the flow of statement execution as being a straight line through the program listing. Although linear flow seems very elementary, always remember that this is the computer's normal mode of operation. Even experienced programmers are sometimes embarrassed to discover that a "bug" in their program was caused by the simple incrementing of the program counter into the wrong portion of the program.

As stated in the introduction of this chapter, a computer would be little more than a glorified adding machine if it were limited to linear flow. There are three general categories of program flow. These are **sequence**, **selection** (conditional execution), and **repetition**. In addition to capabilities in all three of these categories, your computer also has a powerful special case of selection, called **event-initiated branching**. The rest of this chapter shows how to use all of these types of program flow and gives suggestions for choosing the type of flow that is best for your application.

Sequence

Linear Flow

The simplest form of sequence is linear flow. The preceding section showed an example of this type of flow. Although linear flow is not at all glamorous, it has a very important purpose. Most operations required of the computer are too complex to perform using one line of BASIC. Linear flow allows many program lines to be grouped together to perform a specific task in a predictable manner. Although this form of flow requires little explanation, keep these characteristics in mind:

- Linear flow involves **no** decision making. Unless there is an error condition, the program lines involved in this type of flow will always be executed in exactly the same order, regardless of the results of or arguments to any expression.
- Linear flow is the default mode of program execution. Unless you include a statement that stops or alters program flow, the computer will always "fall through" to the next higher-numbered line after finishing the line it is on.

Halting Program Execution

One of the obvious alternatives to executing the next line in sequence is not to execute anything. There are three statements that can be used to block the execution of the next line and halt program flow. Each of these statements has a specific purpose, as explained in the following paragraphs.

Chapter 2 defined a main program as a list of program lines with an END statement on the last line. Marking the end of the main program is the primary purpose of the **END** statement. Therefore, a program can contain only one END statement. The secondary purpose of the END statement is stopping program execution. When an END statement is executed, program flow stops and the program moves into the stopped (non-continuable) state.

It is often necessary to stop the program flow at some point other than the end of the main program. This is the purpose of the **STOP** statement. A program can contain any number of STOP statements in any program context. When a STOP statement is executed, program flow stops and the program moves into the stopped (non-continuable) state. Also, if the STOP statement is executed in a subprogram context, the main program context is restored. (Subprograms and context switching are explained in Chapter 6.)

As an example of the use of STOP and END, consider the following program.

```
100  Radius=5
110  Circum=PI*2*Radius
120  PRINT INT(Circum)
130  STOP
140  Area=PI*Radius^2
150  PRINT INT(Area)
160  END
```

When the [RUN] key is pressed, the computer prints `31` on the CRT and the Run Indicator (lower right corner of CRT) goes off. This first press of the RUN key caused linear execution of lines 100 thru 130, with line 130 stopping that execution. If the [RUN] key is pressed again, the same thing

will happen; the program does **not** resume execution from its stopping point in response to a RUN command. However, RUN can specify a starting point. So, execute `RUN 140`. The computer prints `0` and stops. This command caused linear execution of lines 140 thru 160, with line 160 stopping that execution. However, a RUN command also causes a prerun initialization (see Chapter 2 if this is an unfamiliar term) which zeroed the value of the variable `Radius`.

You could try pressing (CONTINUE) in the preceding example, but you will get an error. A stopped program is not continuable. This leads up to the third statement for halting program flow. Replace the STOP statement on line 130 with a PAUSE statement, yielding the following program.

```
100   Radius=5
110   Circum=PI*2*Radius
120   PRINT INT(Circum)
130   PAUSE
140   Area=PI*Radius^2
150   PRINT INT(Area)
160   END
```

Now press (RUN), and the computer prints `31` on the CRT. Then press (CONTINUE), and the computer prints `78` on the CRT. The purpose of the **PAUSE** statement is to **temporarily** halt program execution, leaving the program counter intact and the program in a continuable state. One common use for the PAUSE statement is in program troubleshooting and debugging. This is covered in Chapter 12. Another use for PAUSE is to allow time for the computer user to read messages or follow instructions. Interfacing with a human is covered in greater depth in Chapter 14, but here is one example of using the PAUSE statement in this way.

```
100   PRINT "This program generates a cross-reference"
110   PRINT "printout. The file to be cross-referenced"
120   PRINT "must be an ASCII file containing a BASIC"
130   PRINT "program."
140   PRINT
150   PRINT "Insert the disc with your files on it and"
160   PRINT "press CONTINUE."
170   PAUSE
180   !     Program execution resumes here after CONTINUE
```

Lines 100 thru 160 are instructions to the program user. Since a user will often just load a program and press (RUN), the programmer cannot assume that the user's disc is in place at the start of the program. The instructions on the CRT remind the user of the program's purpose and review the initial actions needed. The PAUSE statement on line 170 gives the user all the time he needs to read the instructions, remove the program disc, and insert the "data disc". It would be ridiculous to use a WAIT statement to try to anticipate the number of seconds required for these actions. The PAUSE statement gives freedom to the user to take as little or as much time as necessary.

When (CONTINUE) is pressed, the program resumes with any necessary input of file names and assignments. Questions such as "Have you inserted the proper disc?" are unnecessary now. The user has already indicated compliance with the instructions by pressing (CONTINUE).

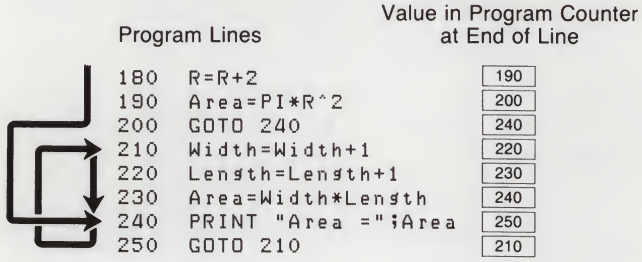
Simple Branching

An alternative to linear flow is branching. Although conditional branching is one of the building blocks for selection structures, the unconditional branch is simply a redirection of sequential flow. The keywords which provide unconditional branching are GOTO, GOSUB, CALL, and FN. The CALL and FN keywords invoke new contexts, in addition to their branching action. This is a complex action that is the topic of an entire chapter (Chapter 6). This section discusses the use of GOSUB and GOTO.

Using GOTO

First, you should be aware that the structuring capabilities available in BASIC make it possible to avoid the use of the unconditional GOTO in most applications. You should also be aware that this is a highly desirable goal. The problem is not anything inherent in the GOTO statement. The problem lies in the programmer's tendency to "glue together" pieces of an algorithm, using more and more GOTOs with each revision. Then comes that inevitable day when a fatal bug reveals that it is impossible to "GET BACK FROM" the last "GO TO". The excessive use of GOTO has been appropriately named **spaghetti coding**. Keep this very descriptive term in mind when you are deciding whether to "just throw something together" or "do it right the first time". (See the section on "Top-Down Design" in Chapter 6.)

The only difference between linear flow and a GOTO is that the GOTO loads the program counter with a value that is (usually) different from the next-higher line number. The GOTO statement can specify either the line number or the line label of the destination. The following drawing shows the program flow and contents of the program counter in a program segment containing a GOTO.



As you can see, the execution is still sequential and no decision making is involved. The first GOTO (line 200) produces a forward jump, and the second GOTO (line 250) produces a backward jump. A forward jump is used to skip over a section of the program. An unconditional backward jump can produce an **infinite loop**. This is the endless repitition of a section of the program. In this example, the infinite loop is line 210 thru 250.

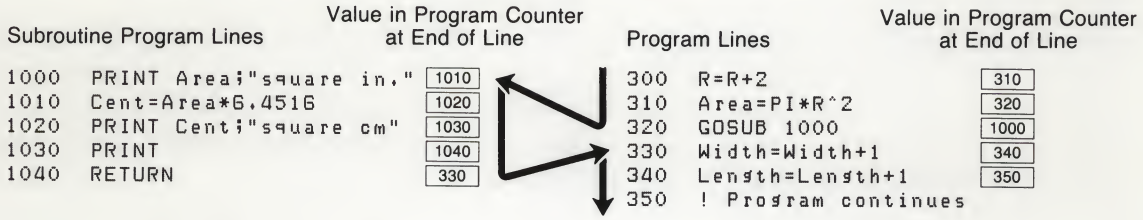
An infinite loop by itself is not usually a desirable program structure. However, it does have its place when mixed with conditional branching or event-initiated branching. Examples of these structures are given later in this chapter.

Using GOSUB

The GOSUB statement is used to transfer program execution to a subroutine. Note that a subroutine and a subprogram are very different in HP BASIC. Calling a **subprogram** invokes a new context. Subprograms can declare formal parameters and local variables. A **subroutine** is simply a

segment of a program that is entered with a GOSUB and exited with a RETURN. Subroutines are always in the same context as the program line that invokes them. There are no parameters passed and no local variables. If you are a newcomer to HP's BASIC, be careful to distinguish between these two terms. They have been used differently in some other programming languages.

The GOSUB is very useful in structuring and controlling programs. The similarity it has to a procedure call is that program flow can automatically return to the proper line when the subroutine is finished. The GOSUB statement can specify either the line label or the line number of the desired subroutine entry point. The following drawing shows the program flow and contents of the program counter in a program segment containing a GOSUB.



Program execution is sequential and no decision making is involved. The main reason that a GOSUB is a more desirable action than a GOTO is the effect of the RETURN statement. The RETURN statement always returns program execution to the line that would have been executed if the GOSUB had not occurred. This is especially useful when using an event-initiated GOSUB. Since it is usually impossible to predict when a user might press a softkey (for example), it is usually impossible to predict what program line should be returned to at the end of a service routine. By using GOSUB and RETURN, the computer does the work for you.

Another common advantage gained from the use of GOSUB is program economy resulting from the consolidation of common tasks. For example, assume that you are writing a page formatter program to neatly print letters, reports, etc. The actions taken at the end of each page might be such things as:

1. Skip two blank lines
2. Print the page number
3. Update the page counter
4. Print a form-feed
5. Zero the line counter

These end-of-page actions might be necessary at many places in the program. For example: in the new-page segment, in the conditional-page algorithm, in the normal line-printing segment, and in the end-of-file process. It would be wasteful duplication to repeat all those end-of-page steps every place they are needed.

That kind of duplication also opens the door to updating problems. Suppose that you wanted to modify the end-of-page action to make it print line-feeds instead of a form-feed for the benefit of a printer that doesn't use form-feeds. If you had duplicated the end-of-page routine in five different places in the program (or was that six?), you will be doing five times as much typing to make the change, and you will probably miss a spot.

The solution is a subroutine. For the sake of completeness in this example, the hypothetical end-of-page subroutine is shown below.

```
540 End_page:  !
550   PRINT USING "2/,K";Pagenumber
560   Pagenumber=Pagenumber+1
570   PRINT CHR$(12);
580   Lines=0
590   RETURN
```

There are no "rules" to say when a program action should be made into a subroutine and when it should be left in linear-flow. The following suggestions may help you decide.

- There is no significant speed penalty for using a subroutine. The time required to process the GOSUB and RETURN is extremely small. If you are having trouble getting your application to run fast enough, it is doubtful that your problems will be solved by removing a couple of GOSUBs. In fact, the resulting loss of "readability" may actually make it more difficult to identify and correct the real problem in timing.
- The "cross-over point" in line overhead is a subroutine that is only three lines long and is called from only two places in the program. In other words, it takes the same number of program lines to duplicate three lines as it does to stick a RETURN on the end of them and add two GOSUB statements. However, there is nothing "magical" about this observation. It does not mean that you shouldn't have a subroutine shorter than three lines, or that you should go around making a subroutine out of every three-line sequence you see repeated. It should simply make you aware of possible improvements that could be made if you see the same sequence repeated in several places in your program.
- Decisions about subroutines are best made on a conceptual level. Although there is nothing wrong with accidentally discovering that you repeated ten lines which would make a good subroutine, it is better to identify the appropriateness of subroutines during planning. One question to ask yourself is, "Does it make sense to handle this task in a subroutine?" If it takes a dozen flags and status variables to select all the variations that are needed from one call to the next, a subprogram is probably a cleaner solution. Lines of code that "just happen" to be repeated in several places are not good candidates for a subroutine. A subroutine should have some identifiable task, like opening a file, normalizing a variable, processing an end-of-page, decoding a keypress, parsing a string, and so forth.

Selection

The heart of a computer's decision-making power is the category of program flow called **selection**, or **conditional execution**. As the name implies, a certain segment of the program either is or is not executed according to the results of a test or condition. This is the basic action which gives the computer an appearance of possessing intelligence. Actually, it is the intelligence of the programmer which is remembered by the program and reflected in the pattern of conditional execution.

Consider a chemistry lab application as an example. There would be little use for a computer whose only function was to turn on a valve when a technician pressed the "START" button. The technician might just as well turn the valve himself. However, if the computer turned on a valve when the "START" was pressed and turned off the valve when a specified pH level occurred, then it is performing a much more useful task. If the example is extended to include state-of-the-art remote-control valves and electronic pH measuring devices, the computer is now significantly outperforming the technician. In this example, (in spite of any fancy instrumentation) the quality that moved the computer from "useless" to "useful" was its ability to **decide** when to turn off the valve. It was the programmer (you) who actually specified the criteria for the decision. Those criteria were then communicated to the computer using conditional-execution program structures. As a result, the computer was able to repeat the programmer's intention with much greater speed and accuracy than a human.

This section presents the conditional-execution statements according to various applications. The following is a summary of these groupings.

1. Conditional execution of one segment.
2. Conditionally choosing one of two segments.
3. Conditionally choosing one of many segments.

Conditional Execution of One Segment

The basic decision to execute or not execute a program segment is made by the IF...THEN statement. This statement includes a numeric expression that is evaluated as being either true or false. If true (non-zero), the conditional segment is executed. If false (zero), the conditional segment is bypassed. Although the expression contained in an IF...THEN is treated as a Boolean expression, note that there is no "BOOLEAN" data type. Any valid numeric expression is allowed.

The conditional segment can be either a single BASIC statement or a program segment containing any number of statements. The first example shows conditional execution of a single BASIC statement.

```
100  IF Ph>7.7 THEN OUTPUT Valve USING "#,B";0
```

Notice the test (`Ph>7.7`) and the conditional statement (`OUTPUT Valve...`) which appear on either side of the keyword THEN. When the computer executes this program line, it evaluates the expression `Ph>7.7`. If the value contained in the variable `Ph` is 7.7 or less, the expression evaluates to 0 (false), and the line is exited. If the value contained in the variable `Ph` is greater than 7.7, the expression evaluates as 1 (true), and the OUTPUT statement is executed. If you don't already understand logical and relational operators, refer to Chapter 4 (numbers) or Chapter 5 (strings).

By the way, the image specifier `#,B` causes the output of a single byte. In the example, the value for that byte is specified as zero (all bits cleared). Presumably, this turns off all devices connected to a GPIO interface. That interface is specified by the value contained in the device selector `Value`. It is beyond the scope of this manual to explain the details of controlling valves and instruments. If you want to do this kind of control, refer to the *BASIC Interfacing Techniques* manual and study the appropriate Interface Installation manual.

The same variable is allowed on both sides of an IF...THEN statement. For example, the following statement could be used to keep a user-supplied value within bounds.

```
IF Number>9 THEN Number=9
```

When the computer executes this statement, it checks the initial value of `Number`. If the variable contains a value less than or equal to nine, that value is left unchanged, and the statement is exited. If the value of `Number` is greater than nine, the conditional assignment is performed, replacing the original value in `Number` with the value nine.

Prohibited Statements

Certain statements are not allowed as the conditional statement in a single-line IF...THEN. The disallowed statements are used for various purposes, but the "common denominator" is that the computer needs to find them during prerun as the first keyword on a line. (A possible exception to this reasoning is REM, which is not allowed because it makes no sense to allow it. Comments certainly aren't executed conditionally. If comments are necessary on an IF...THEN line, the exclamation point can be used.) The following statements are not allowed in a single-line IF...THEN.

Keywords used in the declaration of variables:

COM	OPTION BASE
DIM	REAL
INTEGER	

Keywords that define context boundaries:

DEF FN	FNEND
SUB	SUBEND
END	

Keywords that define program structures:

CASE	FOR
CASE ELSE	IF
ELSE	LOOP
END IF	NEXT
END LOOP	REPEAT
END SELECT	SELECT
END WHILE	UNTIL
EXIT IF	WHILE

Keywords used to identify lines that are literals:

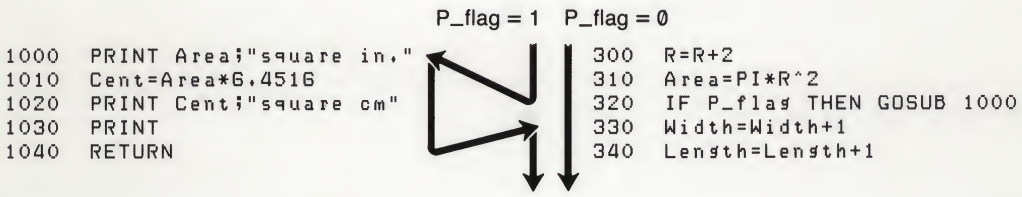
DATA
REM

Conditional Branching

Powerful control structures can be developed by using branching statements in an IF...THEN. Here are some examples.

```
110  IF Free_space<100 THEN GOSUB Expand_file
120  !  The line after is always executed
```

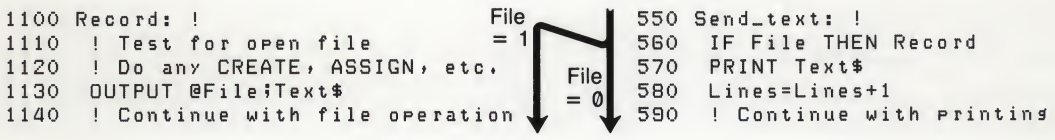
This statement checks the value of a variable called `Free_space`, and executes a file-expansion subroutine if the value tested is not large enough. The same technique can be used with a CALL statement to invoke a subprogram conditionally. One important feature of this structure is that the program flow is essentially linear, except for the conditional "side trip" to a subroutine and back. This is illustrated in the following drawing.



The conditional GOTO is such a commonly used technique that the computer allows a special case of syntax to specify it. Assuming that line number 200 is labeled "Start", the following statements will all cause a branch to line 200 if X is equal to 3.

```
IF X=3 THEN GOTO 200
IF X=3 THEN GOTO Start
IF X=3 THEN 200
IF X=3 THEN Start
```

When a line number or line label is specified immediately after `THEN`, the computer assumes a GOTO statement for that line. (This improves the readability of programs, because phrases like "then start" sound more like English and less like computer jargon.) If execution is redirected by a conditional GOTO (implied or expressed), the program flow does not automatically return to the line following the IF...THEN. Thus, a conditional GOTO acts like a switch on a railroad track. This is illustrated in the following drawing.



Multiple-Line Conditional Segments

If the conditional program segment requires more than one statement, a slightly different structure is used. Let's expand the valve-control example.

```
100  IF Ph>7.7 THEN
110    OUTPUT Valve USING "#,B";0
120    PRINT "Final Ph =";Ph
130    GOSUB Next_tube
140  END IF
150  !  Program continues here
```

Any number of program lines can be placed between a `THEN` and an `END IF` statement. In executing this example, the computer evaluates the expression `Ph>7.7` in the IF...THEN statement. If the result is false, the program counter is set to 150, and execution resumes with the line following the END IF statement. If the condition is true, the program counter is set to 110, and the three conditional statements (lines 110, 120, 130) are executed. Program flow then picks up at line 150, because the END IF is only used during prerun.

When using multiple-line IF...THEN structures, remember to mark the end of the structure with an END IF statement and don't put any of the statements on the same line as the IF...THEN. If the beginning and end of the structure are not properly marked, the computer reports error 347 during prerun.

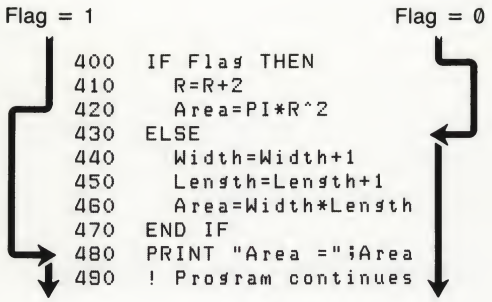
The conditional segment can contain any statement except one which is use to set context boundaries (such as END or DEF FN). In the previous example, the `GOSUB Next_tube` could have been a `GOTO Next_tube`. In that case, program execution does not pass through 150 when the condition is true. A false condition would cause a branch to line 150, while a true condition would send execution from line 100, to 110, to 120, to 130, and then to the line labeled "Next_tube".

If structuring statements are used within a multiple-line IF...THEN, the entire structure must be contained in one conditional segment. This is called **nested** constructs. The following example shows some properly nested constructs. Notice that the use of indenting improves the readability of the code.

```
1000  IF Flag THEN
1010    IF End_of_page THEN
1020      FOR I=1 TO Skip_length
1030        PRINT
1040        Lines=Lines+1
1050      NEXT I
1060    END IF
1070  END IF
```

Choosing One of Two Segments

Often you want a program flow that passes through only one of two paths depending upon a condition. This type of decision is represented pictorally by the following diagram. If you have ever been forced to program this type of structure using only the conditional GOTO, you know that the result is much more confusing than it needs to be.



This language has an IF...THEN...ELSE structrure which makes the one-of-two choice easy and readable. The following example looks at a device selector which may or may not contain a primary address. The variable `Isc` is needed later in the program and must be only an interface select code. If the operator-supplied device selector is greater than 31, the interface select code is extracted from it. If it is equal to or less than 31, it already is an interface select code. (This example assumes that no secondary addressing is used.)

```
500  IF Select>31 THEN
510    Isc=Select DIV 100
520  ELSE
530    Isc=Select
540  END IF
```

Notice that this structure is similar to the multiple-line IF...THEN shown previously. The only difference is the addition of the keyword ELSE. Like the previous example, the structure is terminated by END IF, and the proper nesting of other structures is allowed. The next example shows a program segment that removes certain "escape sequences" from a string. The number of bytes in the escape sequence varies, but can be determined by inspecting the characters following the escape code. Notice the nesting of structures and the conditional branch. When no more escape sequences remain in the string, program execution continues at `Next_seq`.

```
3800 Escape:  !
3810  Point=POS(A$,Esc$)
3820  IF NOT Point THEN Next_seq
3830  IF A$[Point+1;1]<>"&" THEN
3840    A$[Point]=A$[Point+2]       ! 2-byte sequence
3850  ELSE
3860    IF A$[Point+2;1]="d" THEN
3870      A$[Point]=A$[Point+4]     ! 4-byte sequence
3880    ELSE
3890      A$[Point]=A$[Point+5]     ! 5-byte sequence
3900    END IF
3910  END IF
3920  GOTO Escape                   ! Look for more
3930  !
3940 Next_seq:                     ! Program continues here
```

Choosing One of Many Segments

Using SELECT Constructs

Consider as an example the processing of readings from a voltmeter. In this example, we assume that the reading has already been entered, and it contained a function code. These hypothetical function codes identify the type of reading and are shown in the following table.

Function Code	Type of Reading
DV	DC Volts
AV	AC Volts
DI	DC Current
AI	AC Current
OM	Ohms

The first example shows the use of the SELECT construct. The function code is contained in the variable `Funct$`. For the sake of simplicity, the example does not show any actual processing. Comments are used to identify the location of the processing segments. The rules about illegal statements and proper nesting are the same as those discussed previously in the IF...THEN section.

```
2000  SELECT Funct$
2010  CASE "DV"
2020    !
2030    ! Processing for DC Volts
2040    !
2050  CASE "AV"
2060    !
2070    ! Processing for AC Volts
2080    !
2090  CASE "DI"
2100    !
2110    ! Processing for DC Amps
2120    !
2130  CASE "AI"
2140    !
2150    ! Processing for AC Amps
2160    !
2170  CASE "OM"
2180    !
2190    ! Processing for Ohms
2200    !
2210  CASE ELSE
2220    BEEP
2230    PRINT "INVALID READING"
2240  END SELECT
2250  ! Program execution continues here
```

Notice that the SELECT construct starts with a SELECT statement specifying the variable to be tested and ends with an END SELECT statement. The anticipated values are placed in CASE statements. Although this example shows a string tested against simple literals, the SELECT statement works for numeric or string variables or expressions. The CASE statements can contain constants, variables, expressions, comparison operators, or a range specification. The anticipated values, or **match items**, must be of the same type (numeric or string) as the tested variable.

The CASE ELSE statement is optional. It defines a program segment that is executed if the tested variable does not match any of the cases. If CASE ELSE is not included and no match is found, program execution simply continues with the line following END SELECT.

The following example shows a numeric variable tested with comparison operators and a range specifier.

```
1500  SELECT Ds
1510  CASE <1
1520    ! Processing for invalid device selector
1530  CASE 1 TO 31
1540    ! Processing for interface select code
1550  CASE >31
1560    ! Contains primary address
1570  END SELECT
```

A CASE statement can also specify multiple matches by separating them with commas, as shown below.

```
CASE -1,1,3 TO 7,>15
```

The following CASE statement shows the use of a string expression, rather than a simple constant.

```
CASE CHR$(27)&")@"&Eol$
```

You should be aware that if an error occurs when the computer tries to evaluate an expression in a CASE statement, the error is reported for the line containing the SELECT statement. This is a result of the nature of SELECT constructs and is not a bug. However, it can make things a bit confusing if you aren't aware of it. An error message pointing to a SELECT statement actually means that there was an error in that line **or** in one of the CASE statements. It requires more "detective work" on your part to locate the line which actually contains the erroneous expression.

Using the ON Statement

This type of program flow can also be generated with the ON statement and some additional processing. Let's do a string example first, using the previous voltmeter example. All the anticipated values are placed in a simple string. This string is then searched using the POS function. The results of the POS function are adjusted to become consecutive integers beginning with one. This result can then be used in the ON statement.

```
100  Match$="DVAVDIAIOM"
.
.
.
500  Pointer=POS(Match$,Funct$)
510  Pointer=INT((Pointer-1)/2+1)
520  ON Pointer+1 GOSUB Case_else,Case_dv,Case_av,
Case_di,Case_ai,Case_om
```

Notice that a match can only cause values of 1, 3, 5, 7, or 9 from the POS function. A "match not found" gives a value of 0. Line 510 converts these to consecutive integers from 0 thru 5. The `Pointer+1` expression in line 520 shifts the values to a range 1 thru 6, which is acceptable to the ON statement.

The values of the match characters will determine the "pre-processing" necessary. If you are trying to match single bytes, simply adding one to the results of the POS is all that is necessary. Finding 3-letter sequences requires a line like 510, only with a division by 3. Note also that, except for single bytes, this method may not always work. For example, if the current ranges had been indicated by DA and AA (instead of DI and AI), Match$ would be "DVAVDAAAOM". A subsequent search for "AA" would return 6 instead of 7 — not good. In a case like that, there are two choices. One approach is to rearrange the string being searched; "DVAVDAOMAA" would work. Perhaps the items in the string could be separated with a "pad" character and the calculation adjusted accordingly. The other approach is to make each match value a separate element of a string array. The array could then be "searched" with a FOR...NEXT loop. This approach works well to resolve conflicts, especially with long match strings. However, the extra code lines and array accesses slow the process down significantly.

The ON statement can also be used for numeric values. If the numeric values you are trying to match just happen to be consecutive integers starting with one, the variable to be tested can be used in the ON statement. However, programmers don't usually get that lucky. To match arbitrary values, the following trick can be used. This example tests the three cases: <0, 1, and >1.

```
700  Pointer=1*(X<0)+2*(X=1)+3*(X>1)
710  ON Pointer GOSUB Negative,One,Greater
```

Assuming that you use non-overlapping comparison tests, only one of the values in parentheses will be true. The system returns a value of "1" for true. This is multiplied times the corresponding factor to give the final value to Pointer. All the other factors drop out because their comparison result is zero. Programmers who like strong type checking may raise an eyebrow at this technique, but it works.

Another way of testing for numbers that are integers between 0 and 255 is to use the CHR$ function to create string bytes and apply the POS function as explained previously.

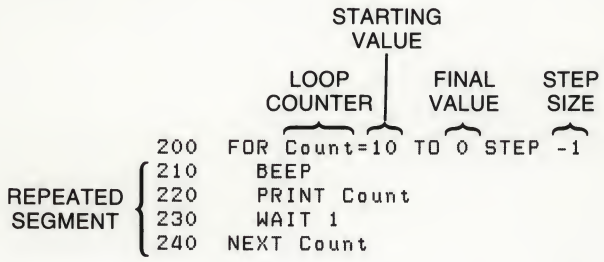
Repetition

Humans usually prefer tasks with variety that avoid tedious repetition. A computer does not have this shortcoming. You have four structures available for creating repetition. The FOR...NEXT structure is used for repeating a program segment a predetermined number of times. Two other structures (REPEAT...UNTIL and WHILE) are used for repeating a program segment indefinitely, waiting for a specified condition to occur. The LOOP...EXIT IF structure is used to create an iterative structure that allows multiple exit points at arbitrary locations.

The FOR...NEXT structure is used for repeating a program segment a predetermined number of times. Two other structures (REPEAT...UNTIL and WHILE) are used for repeating a program segment indefinitely, waiting for a specified condition to occur. The LOOP...EXIT IF structure is used to create an iterative structure that allows multiple exit points at arbitrary locations.

Fixed Number of Iterations

The general concept of repetitive program flow can be shown with the FOR...NEXT structure. With this structure, a program segment is executed a predetermined number of times. The FOR statement marks the beginning of the repeated segment and establishes the number of repetitions. The NEXT statement marks the end of the repeated segment. This structure uses a numeric variable as a **loop counter**. This variable is available for use within the loop, if desired. The following drawing shows the basic elements of a FOR...NEXT loop.



The number of loop iterations is determined by the FOR statement. This statement identifies the loop counter, assigns a starting value to it, specifies the desired final value, and determines the step size that will be used to take the loop counter from the starting value to the final value. When the loop counter is an INTEGER, the number of iterations can be predicted using the following formula:

$$\text{INT}\left(\frac{\text{Step Size} + \text{Final Value} - \text{Starting Value}}{\text{Step Size}}\right)$$

Note that the formula applies to the values in the variables, not necessarily the numbers in the program source. For example, if you use an INTEGER loop counter and specify a step size of 0.7, the value will be rounded to one. Therefore, 1 should be used in the formula, not 0.7.

The loop counter can be a REAL number, with REAL quantities for the step size, starting, or final values. In some cases, using REAL numbers will cause the number of iterations to be off by one from the preceding formula. This is because the NEXT statement performs an "increment and compare", and there is a slight inaccuracy in the comparison of REAL numbers. If you are interested, this is discussed in the next chapter. However, there is no "clean" way around it with FOR...NEXT loops. Here is an example:

```
200  Count=0
210  FOR X=10 TO 20
220    Count=Count+1
230    PRINT Count
240  NEXT X
```

According to the formula, this loop should execute 11 times: INT((1 + 20 − 10)/1 = 11). The result on the CRT confirms this when the loop is executed. If line 210 is changed to:

```
210  FOR X=1 TO 2 STEP .1
```

the formula still yields 11 as the number of iterations. However, executing the loop produces only 10 repetitions. This is because of a very, very small accumulated error that results from the successive addition of one-tenth. The error is less significant than the 15th digit, but discernable to the computer. In this case, rounding cannot be performed at a time that would help. When you find yourself in this situation, one solution is to add a slight adjustment factor to the final value. One half of the step size is a convenient adjustment factor. The following line does give the 11 iterations predicted by the formula.

```
210  FOR X=1 TO 2.05 STEP .1
```

Remembering the "increment and compare" operation at the bottom of the loop is helpful. After the loop counter is updated, it is compared to the final value established by the FOR statement. If the loop counter has **passed** the specified final value, the loop is exited. If it has **not passed** the specified final value, the loop is repeated. The loop counter retains its exit value after the loop is finished. This is not necessarily one full step past the final value. For example:

```
FOR I=1 TO 9.9
```

This statement establishes a loop that executes nine times (the default step size is one). The variable I has the value 10 when the loop is exited.

```
FOR Count=12 TO 1 STEP -0.3
```

This statement establishes a loop that executes 37 times. The variable Count has the value .9 when the loop is exited. Notice that negative step sizes are allowed using the same keywords as positive step sizes.

The final points to mention concern the execution of the FOR statement. If any variables are present to the right of the equal sign, the value used is the value they have when the FOR statement is executed. Remember that the FOR statement is only executed once before the loop begins. Also, if the number of iterations evaluates to zero or less, the loop is not executed and program execution goes immediately to the line following the NEXT statement. Here are some examples.

```
400  FOR Item=First TO Last
410    GOSUB Process
420    Last=Last+1
430  NEXT Item
440  ! Execution continues here
```

This loop would not be executed if Last were less than First. This is almost always desirable, since it prevents the subroutine Process from being invoked with a null item. Also notice that the number of iterations is fixed at loop entry when line 400 is executed. That number of iterations does not change when the value of Last is changed.

```
FOR Item=Item+1 TO Last
```

The variable Item is used as the loop counter. It receives a starting value that is one greater than the value it had when this line is executed.

Conditional Number of Iterations

The FOR...NEXT loop produces a fixed number of iterations, established by the FOR statement before the loop is executed. Some applications need a loop that is executed until a certain condition is true, without specifically stating the number of iterations involved. Consider a very simple example. The following segment asks the operator to input a positive number. Presumably, negative numbers are not acceptable. A looping structure is used to repeat the entry operation if an improper value is given. Notice that it is not important **how many times** the loop is executed. If it only takes once, that is just fine. If the poor operator takes ten tries before he realizes what the computer is asking for, so be it. What is important is that a **specific condition** is met. In this example, the condition is that a value be non-negative. As soon as that condition has been satisfied, the loop is exited.

```
800  REPEAT
810    INPUT "Enter a positive number",Number
820  UNTIL Number>=0
```

A typical use of this is an iterative problem involving non-linear increments. One example is musical notes. Performing the same operation on all the notes in a 3-octave band is a repetitive process, but not a linear one. Musical notes are related geometrically by the 12th root of two. The following example simply prints the frequencies involved, but your application could involve any number of operations.

```
1200  Note=110        ! Start at low A
1210  REPEAT
1220    PRINT Note;
1230    Note=Note*2^(1/12)
1240  UNTIL Note>880   ! End at high A
```

For this example, a FOR...NEXT loop might have been used, with the loop counter appearing in an exponent. That would work because it is relatively easy to know how many notes there are in three octaves of the musical scale. However, the REPEAT...UNTIL structure is more flexible than FOR...NEXT when working with exponential data in general. Examples often occur in the area of graphics. The following segment could be used to plot audio frequency data, where the x-axis is logrithmic.

```
1500  Freq=20
1510  MOVE LOG(Freq),FNFunction(Freq)
1520  REPEAT
1530    DRAW LOG(Freq),FNFunction(Freq)
1540    Freq=Freq*1.2
1550  UNTIL Freq>20000
```

The flexibility of this structure is in line 1540. By increasing the frequency with a factor of 1.2, a very fast but rough graph is generated. (You need the GRAPH BIN to run this example.) This lets you place axes, labels, markers, etc. where you want them without waiting for a time-consuming plot for each cosmetic change. Once you have the desired appearance, you could change line 1540 to `Freq=Freq*1.01`. This would greatly increase the resolution of the plot (and reduce its speed). To take it one step further, you could make the "resolution factor" a variable and input its value at the start of the program. That would make it easy to try many different increments to achieve the best compromise between resolution and smoothness. Attempting a similar technique with FOR...NEXT loops would involve many extra (and unnecessary) calculations.

The WHILE loop is used for the same purpose as the REPEAT loop. The only difference between the two is the location of the test for exiting the loop. The REPEAT loop has its test at the bottom. This means that the loop is always executed at least once, regardless of the value of the condition. The WHILE loop has its test at the top. Therefore, it is possible for the loop to be skipped entirely (if the conditions so dictate). The following segment shows the same plotting example using a WHILE loop.

```
1500  Freq=20
1510  MOVE LOG(Freq),FNFunction(Freq)
1520  WHILE Freq<=20000
1530    DRAW LOG(Freq),FNFunction(Freq)
1540    Freq=Freq*1.2
1550  END WHILE
```

The next segment shows the use of conditional branching to simulate a REPEAT...UNTIL structure.

```
1500  Freq=20
1510  MOVE LOG(Freq),FNFunction(Freq)
1520 Loop_top:  !
1530    DRAW LOG(Freq),FNFunction(Freq)
1540    Freq=Freq*1.2
1550  IF Freq<=20000 THEN Loop_top
```

The WHILE structure can also be simulated using GOTO statements. The following segment shows this technique.

```
1500  Freq=20
1510  MOVE LOG(Freq),FNFunction(Freq)
1520 Loop_top:  !
1530  IF Freq>20000 THEN Loop_exit
1540    DRAW LOG(Freq),FNFunction(Freq)
1550    Freq=Freq*1.2
1560  GOTO Loop_top
1570 Loop_exit:  !
```

The REPEAT...UNTIL and WHILE structures are especially useful for tasks that are impossible with a FOR...NEXT loop. One such situation is a loop where both the loop counter and the final value are changing. Consider the example of stripping all control characters from a string. This can't be done in a loop that starts `FOR I=1 TO LEN(A$)`, because the length of A$ changes each time a character is deleted. Therefore, the loop counter used as a subscript will eventually exceed the length of the string by more than one, generating an error. The WHILE loop does not have this problem. Note that the test at the top of the loop prevents the subscripting from being attempted on a null string. This is necessary to avoid an error.

```
600  I=1
610  WHILE I<=LEN(A$)
620    IF A$[I;1]<CHR$(32) THEN
630      A$[I]=A$[I+1]
640    ELSE
650      I=I+1
660    END IF
670  END WHILE
```

Arbitrary Exit Points

A pass through any of the loop structures discussed so far included the entire program segment between the top and the bottom of the loop. There are times when this is not the desired program flow. The LOOP structure defines a repeated program segment and allows any number of conditional exits points in that segment.

For the first example, consider a search and replace operation on string data. In this example, the "shift out" control character is being used to initiate underlining on a printer that understands standard escape sequences. The "shift in" control character is used to turn off the underline mode. (There is nothing significant about this choice of characters. Any combination of characters could serve the same purpose.)

One approach is to use a loop to search every character in every string to see if it is one of the special characters. There are two problems with this method. First, it is a little cumbersome when the replacement string is a different length than the target string. Second, it is slow. Admittedly, speed it not a significant consideration when driving common mechanical printers. But the destination might eventually be a lazer printer or mass storage file, making the program's speed more visible.

A better approach is to use the POS function to locate the target string. Since this function locates only the first occurrence of a pattern, it must be placed in a loop to insure that multiple occurrences will be found. The LOOP structure is well suited to this task, as shown in the following example.

```
2000  LOOP
2010    Position=POS(A$,CHR$(14))
2020  EXIT IF NOT Position
2030    A$[Position]=CHR$(27)&"&dD"&A$[Position+1]
2040  END LOOP
2050  !
2060  LOOP
2070    Position=POS(A$,CHR$(15))
2080  EXIT IF NOT Position
2090    A$[Position]=CHR$(27)&"&d@"&A$[Position+1]
2100  END LOOP
2110  ! Last EXIT goes to here
```

In this segment, all occurrences of "shift out" are replaced by "escape &dD" to enable underline mode. All occurrences of "shift in" are replaced by "escape &d@" to disable underlining. Notice that there is no problem replacing one character with four (assuming that A$ is large enough). Lines containing no special characters are processed by only two POS functions, which is much faster and cleaner than performing two comparisons for every character in every line.

Another common use for this structure is the processing of operator input. Recall the REPEAT...UNTIL example that tested for the input of a positive number. Although this structure kept the computer happy, it left the operator in the dark. The LOOP structure provides for the additional processing needed, as shown in the following example.

```
200  LOOP
210    INPUT "Enter a positive number.",Number
220  EXIT IF Number>=0
230    BEEP
240    PRINT
250    PRINT "Negative numbers are not allowed."
260    PRINT "Repeat entry with a positive number.
270  END LOOP
```

Another point to remember is that the LOOP structure permits more than one exit point. This allows loops that are exited on a "whichever comes first" basis. Also, the EXIT IF statement can be at the top or bottom of the loop. This means that the LOOP structure can serve the same purposes as REPEAT...UNTIL and WHILE, if that suits your programming style.

The EXIT IF statement must appear at the same nesting level as the LOOP statement for a given loop. This requirement is best shown with an example. In the "WRONG" example, the EXIT IF statement has been nested one level deeper than the LOOP statement because it was placed in an IF...THEN structure.

WRONG

```
600  LOOP
610    Test=RND-.5
620    IF Test<0 THEN
630      GOSUB Negative
640    ELSE
650      EXIT IF Test>.4
660      GOSUB Positive
670    END IF
680  END LOOP
```

RIGHT

```
600  LOOP
610    Test=RND-.5
620  EXIT IF Test>.4
630    IF Test<0 THEN
640      GOSUB Negative
650    ELSE
660      GOSUB Positive
670    END IF
680  END LOOP
```

Event-Initiated Branching

Your computer has a special kind of program flow that provides some very powerful tools. This tool, called **event-initiated branching**, uses interrupts to redirect program flow. The process can be visualized as a special case of selection. Everytime program flow leaves a line, the computer executes an "event-checking" routine. This is set of machine-language "if...then" statements concerning interrupts. If an event is enabled and "true", this "event-checking" routine causes the program to branch.

The process of "event checking" is represented in the following lines. Notice that it is possible for event-initiated branching to occur at the end of any program line, which includes the lines of a subprogram. This can give the appearance of "middle-of-line" branching when it occurs during a user-defined function. subprogram. These potential branching points are marked by the words "gosub event_check". This does not refer to a BASIC subroutine, but is just a symbolic reminder of where event-initiated branching can occur. If the operating system finds a "true" event, a branch is taken. If not, program execution resumes with the "normal" program flow.

```
10  PRINT X( gosub event_check)
20  X=X+1 (gosub event_check)
30  GOTO 10 (gosub event_check)
```

Enabling Events

Event-initiated branching is established by the ON-event statements. Here is a list of the statements that fall in this category:

ON END	ON ERROR	ON CYCLE	ON EOR
ON KBD	ON KEY	ON DELAY	ON EOT
ON KNOB	ON INTR	ON TIME	
ON TIMEOUT	ON SIGNAL		

The ON END event is used to detect when the end of a mass storage file is reached. This is discussed in Chapter 7.

The ON CYCLE, ON DELAY and ON TIME statements are used to direct program flow using the clock. They are discussed in Chapter 9.

The ON ERROR event is used to trap run-time errors and provide for error-recovery routines. This is discussed in Chapter 11.

The ON KBD, ON KEY, and ON KNOB events pertain to various parts of the keyboard and are used to enhance the "human interface" of programs. ON KBD lets you re-define the entire keyboard to suit the needs of your program. This is discussed in the *BASIC Interfacing Techniques* manual. The ON KEY statement is used to define and label the softkeys on the keyboard. The ON KNOB statement lets you capture turns of the knob. This chapter has some examples of ON KEY and ON KNOB.

The ON EOR, ON EOT, ON SIGNAL, ON INTR and ON TIMEOUT events pertain to data transfer, interfaces and I/O operations. These are discussed in the *BASIC Interfacing Techniques* manual.

The best way to understand how event-initiated branches operate in a program is to sit down at the computer and try a few examples. Start by entering the following short program.

```
110  ON KEY 1 LABEL "Inc" GOSUB Plus
120  ON KEY 5 LABEL "Dec" GOSUB Minus
130  !
140 Spin:  DISP X
150  GOTO Spin
160  !
170 Plus:  X=X+1
180  RETURN
190  !
200 Minus:  X=X-1
210  RETURN
220  END
```

Notice the various structures in this sample program. The ON KEY statements are executed only once at the start of the program. Once defined, these event-initiated branches remain in effect for the rest of the program. (Disabling and deactivating are discussed later.) The program segment labeled "Spin" is an infinite loop. If it weren't for interrupts, this program couldn't do anything except display a zero. However, there is an implied IF...THEN at the end of lines 140 and 150 because of the ON KEY action. This allows a selection process to occur. Either the "Plus" or the "Minus" subroutine can be selected as a result of softkey presses. These are normal subroutines terminated with a RETURN statement. (In the context of interrupt programming, these subroutines are called **service routines.**) The following section of pseudocode shows what the program flow of the "Spin" segment actually looks like to the computer.

```
Spin: display X
   if Key1 then gosub Plus
   if Key5 then gosub Minus
   goto Spin
```

This pseudocode is an over-simplification of what is actually happening, but it shows that the "Spin" segment is not really an infinite loop with no decision-making structure. Actually, most programs that use event-initiated branching to control program flow will contain what appears to be an infinite loop. That is the easiest way to "keep the computer busy" while it is waiting for an interrupt.

Now run the sample program you just entered. Notice that the bottom of the CRT displays an inverse-video label area. These labels are arranged to correspond to the layout of the softkeys. The labels are displayed when the softkeys are active and are not displayed when the softkeys are not active. Any label which your program has not defined is blank. The label areas are defined in the ON KEY statement by using the keyword "LABEL" followed by a string.

The starting value in the display line is zero, since numeric variables are initialized to zero at prerun. Each time you press [k1] or [f1], the displayed value of X is incremented. Each time you press [k5] for [f5], the displayed value of X is decremented. This simple demonstration should aquaint you with the basic action of the softkeys.

It is possible to make structures that are much more elaborate, with assignable priorities for each key, and keys that interrupt the service routines of other keys. There are many applications where priorites are not of any real significance, such as the example program running now. However, priorities will sometimes cause unexpected flow problems. One type of priority problem can be shown with a simple modification to our example program. Insert the following line right after line 170.

```
171  GOTO 171
```

Now run the program and press [k1] or [f1]. Notice that the program "locks up" and all subsequent presses of either softkey do nothing. This is not simply because line 171 creates an infinite loop. The program segment at "Spin" was a infinite loop and that didn't bother the softkeys at all. The problem is that the priority for the "Plus" service routine is higher than the main program priority. None of the softkeys have been assigned a high enough priority to interrupt another service routine. A full discussion on interrupt priority can be found in the "Interface Events" chapter of the *BASIC Interfacing Techniques* manual. If you think you have an application that is "priority sensitive", read that section carefully.

Using the Knob

One characteristic of interrupt-driven program flow is that the computer's decisions can be more easily synchronized with the actions of devices connected to it. This type of application is often called **real-time programming**. An important example of real-time programming is machine control. A computer running an automatic packing machine must turn off the flow immediately when the jar is full. It is not acceptable for the computer to wait until the inventory printout is done and peanut butter is dumped all over the conveyor belt. Although machine control applications are very important, their extensive interfacing makes them inconvenient or impossible to use as demonstration programs in a manual such as this.

Another common example of real-time programming is computer games. The computer is expected to respond "instantly" to button presses, lever movement, etc. The operator expects immediate correlation between their input and the computer's output or display. Your BASIC Utilities Disc has a couple of simple games on it that demonstate interaction between the CRT, softkeys, and knob. Feel free to list any of the programs on that disc if you want further examples of various techniques.

The following program is a very short example that demonstates a real-time interaction between the knob and the CRT. If you run this example program and turn the knob, you will see the kind of interaction that might be used for cursor control in a text editor. Obviously, a real cursor-control routine would be much more sophisticated, but this demonstrates the basic idea.

```
10      ON KNOB .1 GOSUB Move_blip
20 Spin:      GOTO Spin
30      !
40 Move_blip:  !
50      PRINT TABXY(Spotx,Spoty);" ";
60      Spotx=Spotx+KNOBX/5
70      Spoty=Spoty+KNOBY/5
80      IF Spoty<1 THEN Spoty=1
90      IF Spoty>18 THEN Spoty=18
100     IF Spotx<1 THEN Spotx=1
110     IF Spotx>50 THEN Spotx=50
120     PRINT TABXY(Spotx,Spoty);CHR$(127);
130     RETURN
140     END
```

This example uses a short infinite loop to wait for pulses from the knob (line 20). Interrupts from the knob are enabled by the ON KNOB statement in line 10. The service routine erases the old "blip", performs some scaling and range checking on the knob input, and prints the new "blip".

The scaling and range checking are very important in this kind of interactive routine. Humans expect their interface to have a certain "feel". Displays should not change too quickly or too slowly. Certain kinds of displays are expected to change logarithmically, others are expected to change linearly. The boundary values of variables are expected to conform to the boundaries of the display. To initiate yourself to some of these concepts, try modifying this simple example. Remove one or more of the range checking lines. (An easy way to do this kind of editing is to place an exclamation point in front of the statement. This turns it into a comment, removing it from the flow of execution. But it can be easily returned to the program by deleting the exclamation point.) Also try changing the scaling factor in lines 60 and 70. Notice the "feel" that results from larger and smaller increments, or even logarithmic scaling.

Deactivating Events

Knowing how to "turn off" the interrupt mechanism is just as important as knowing how to enable it. Often, an event is a desired input during one part of the program, but not during another. You might use softkeys to set certain process parameters the start of a program, but you don't want interrupts from those keys once the process starts. For example, a report generating program could use a softkey to select single or double spacing. This key should be disabled once the printout starts so that an accidental keypress does not cause the computer to abort the printout and return to the questions at the beginning of the program. On the other hand, you might want an "Abort" key that does precisely that. The important thing is that you decide on the desired action and make the computer obey your wishes.

Before going any further, let's explain some important terminology. There are two general methods for "turning off" an interrupt. If an interrupt source is **deactivated**, it no longer has any influence on program flow. You can press a deactivated key all day long and nothing will happen. However, if an event is **disabled**, its action has only been temporarily postponed. The computer remembers that a key was pressed while it was disabled, and the action for that key will occur at the earliest opportunity once the disabled state is removed. There are examples in this section to demonstate the difference betweem these two conditions.

All the "ON-event" statements have a corresponding "OFF-event" statement. This is one way to deactivate an interrupt source.

- OFF KEY deactivates interrupts from the softkeys. If a softkey is pressed while deactivated, it does nothing.
- OFF KNOB deactivates the ON KNOB interrupts. Turning the knob while ON KNOB is deactivated causes normal scrolling on the CRT.

The following example shows one use of OFF KEY to disable the softkeys. (Note that [kn] is used in the description. If you have an HP 46020A keyboard, just substitue [fn].) A softkey is used to select a parameter for a small printing routine. Each press of [k1] increments and displays the step size that will be used as an interval between the printed numbers. When the desired step size has been selected, [k4] is pressed to start the printout. Enter and run this example. Notice that with line 240 and 250 commented out, the **softkey menu**, or label area, never changes.

```
100 Begin:  !
110   ON KEY 1 LABEL " DELTA" GOSUB Step_size
120   ON KEY 4 LABEL " START" GOTO Process
130   Inc=1
140   DISP "Step Size = 1"
150   !
160 Spin: GOTO Spin              ! Wait for keypress
170   !
180 Step_size:  !
190   Inc=Inc+1                  ! Change increment
200   DISP "Step Size =";Inc
210   RETURN
220   !
230 Process:  !
240 ! OFF KEY                    ! Deactivate first choices
250 ! ON KEY 8 LABEL " ABORT" GOTO Leave
260   Number=0
270   FOR I=1 TO 10
280     Number=Number+Inc
290     PRINT Number;
300     WAIT .6
310   NEXT I
320 Leave:  !
330   OF KEY 8                  ! Deactivate ABORT
340   PRINT                      ! End line
350   GOTO Begin                 ! Start over
360   END
```

Now run the example again and press [k1] or [k4] while the printout is in progress. Notice that the keys are still active and produce undesired effects on the printing process. To "fix this bug", remove the exclamation point from line 240. This disables all the softkeys when the printing process starts. Notice that the softkey menu goes away when no sofkeys are active. This is a very handy feature while you are experimenting with interupts. It provides immediated feedback to indicate when interrupts are active and when they are not.

Finally, remove the exclamation point from line 250. Now, the softkey menu appears during the printing process. However, the choices are different than at the start of the program. The keys used to select the parameter and start the process are not active, because they are not needed at this point in the program. Instead, [k8] can be used to "gracefully" abort the process and return to the start of the program.

The OFF KEY statement can include a key number to deactivate a selected key. This was done in line 330.

Disabling Events

All the previous examples have shown complete deactivation of the softkeys. It is also possible to temporarily disable an event-initiated branch. This is done when an active event is desired in a process, but there is a special section of the program that you don't want to be interrupted. Since it is impossible to predict when an external event will occur, the special section of code can be "protected" with a DISABLE statement. This is sometimes necessary to prevent a certain variable from being changed in the middle of a calculation or to insure that a interface polling sequence runs to completion. It is difficult in a short, simple example to show **why** you would need to do this. But it is not difficult to show **how** to do it.

```
100    ON KEY 9 LABEL " ABORT" GOTO Leave
110    !
120 Print_line:  !
130    DISABLE
140    FOR I=1 TO 10
150      PRINT I;
160      WAIT .3
170    NEXT I
180    PRINT
190    ENABLE
200    GOTO Print_line
210    !
220 Leave: END
```

This example shows a DISABLE and ENABLE statement used to "frame" the Print_line segment of the program. The "ABORT" key is active during the entire program, but the branch to exit the routine will not be taken until an entire line is printed. The operator can press the "ABORT" key at any time. The keypress will be **logged**, or remembered, by the computer. Then when the ENABLE statement is executed, the event-initated branch is taken. Enter and run the example to observe this method of delaying interrupt servicing.

Notes

Chapter 4 Numeric Computation

Introduction

When most people think about computers, the first thing that they think of is number-crunching, the giant calculator with a brain. Whether this is an accurate impression or not, numeric computations are an important part of computer programming.

Numeric computations deal exclusively with numeric values. Thus, adding two numbers and finding a sine or a logarithm are all numeric operations; while converting bases and converting a number to a string or a string to a number are not. (Converting bases and converting numbers to strings and strings to numbers are covered in the chapter on String Operations.)

The most fundamental numeric operation is the assignment operation, achieved with the LET statement. The LET statement originally required the keyword LET for BASIC interpreters, but your computer makes it optional. Thus the following statements are equivalent:

```
LET A = A + 1
A = A + 1
```

Numeric Data Types

There are two numeric data types in BASIC that you are using, INTEGER and REAL. Any numeric variable that is not declared an INTEGER is a REAL variable. The valid range for REAL numbers is approximately:

$-1.797\ 073\ 134\ 862\ 315 \times 10^{308}$ thru $1.797\ 073\ 134\ 862\ 315 \times 10^{308}$

the smallest non-zero REAL value allowed is approximately:

$\pm\ 2.225\ 073\ 858\ 507\ 202 \times 10^{-308}$

A REAL can also have the value of zero.

An INTEGER can have any whole-number value from:

$-32\ 768$ thru $+32\ 767$

Both REAL and INTEGER variables may be grouped into arrays.

Declarations

It is good programming practice to declare all variables, and both INTEGER and REAL statements are provided for declaring variables:

```
INTEGER I, J, Days(5), Weeks(5:17)
REAL X, Y, Voltage(4), Hours(5,8:13)
```

Each of the above statements declares two scalar and two array variables. A scalar is a variable which can, at any given time, represent a single value. An array is a subscripted variable, and can contain multiple values, accessed by subscripts. It is posible to specify both the lower and upper bounds of an array, or to specify the upper bound only, and use the existing OPTION BASE as the lower bound. Details on declarations of arrays and how to use them are provided later in this chapter when arrays are dealt with in detail. The DIM statement may also be used to declare a REAL array.

```
DIM R(4,5)
```

An ALLOCATE statement can be used to declare both REAL and INTEGER arrays.

```
ALLOCATE REAL Co_ords(2,1:Points), INTEGER Status(1:Points)
```

The ALLOCATE statement allows you to dynamically allocate memory in programs which need tight control over memory use.

Type Conversions

The computer will automatically convert between REAL and INTEGER values in assignment statements and when parameters are pased by value in function and subprogram calls. When parameters are passed by reference the conversion will not be made and a type mismatch error will be reported. Whenever numbers are converted from REAL to INTEGER representations, information can be lost. There are two potential problem areas in this conversion, rounding errors and range errors.

The computer will automatically convert between types when an assignment is made, and this presents no problem when an INTEGER is converted to a REAL. However, when a REAL is converted to an INTEGER, the REAL is rounded to the closest INTEGER value. When this is done, all information about the number to the right of the radix (decimal point) is lost. If the fractional information is truly not needed, there is no problem, but converting back to a REAL will not reconstruct the lost information — it stays lost.

Another potential problem with REAL to INTEGER conversions is the difference in ranges. While REAL values range from approximately -10^{308} to $+10^{308}$, the INTEGER range is only from $-32\,768$ to $+32\,767$ (approximately -10^4 thru $+10^4$). Obviously, not all REAL values can be rounded into an equivalent INTEGER value. This problem can generate INTEGER Overflow errors.

While the rounding problem is important, it does not generate an execution error. The range problem **can** generate an execution error, and you should protect yourself from crashing the program by either testing values before assignments are made, or by using ON ERROR to trap the error, and making corrections after the fact.

The following fragment shows a method to protect against INTEGER overflow errors:

```
200  IF X > 32767 THEN X= 32767
210  IF X < -32768 THEN X = -32768
220  Intx = X
```

It is possible to achieve the same effect using `MAX` and `MIN` functions:

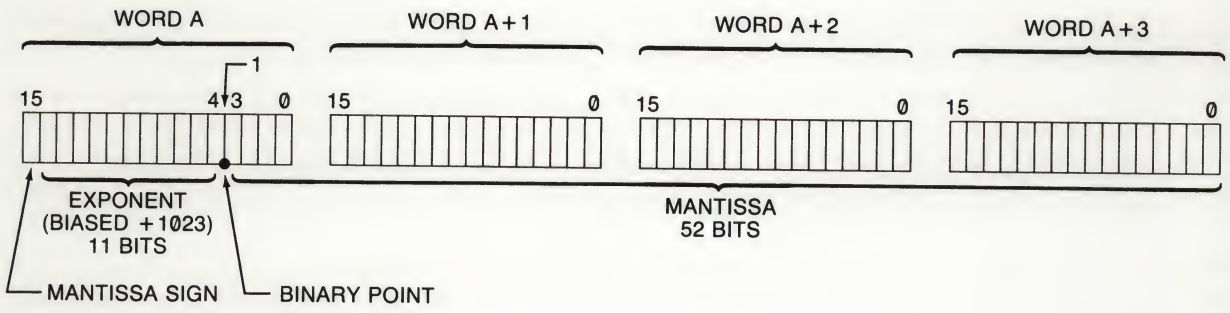
```
200 Y = MAX(MIN(X, 32767), -32768)
```

Both these methods limit the excursion, but lose the fact that the values were originally out of range. If out-of-range is a meaningful condition, an error handling trap is more appropriate.

```
200 IF (-32768<=X) AND (X<=32767) THEN
210    Y = X
220 ELSE
230    GOSUB Out_of_range
240 END IF
```

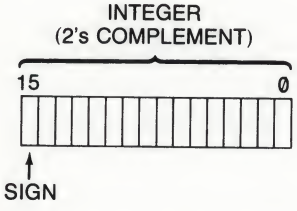
Internal Numeric Formats

The storage format for REAL and INTEGER numbers in memory are as follows:



$$-1^{\text{mantissa sign}} \times 2^{\text{exponent} - 1023} \times 1.\text{mantissa}$$

Storage Format for REAL Variables



Storage Format for INTEGER Variables

Precision and Accuracy: The Machine Limits

Your computer stores all REAL variables with a sign, approximately 15 significant digits, and the exponent value. For most engineering and other applications, rounding errors are not a problem because the resolution of the computer is well beyond the limitations of most scientific measuring devices. However, when high-resolution numerical analysis requires accuracy approaching the limits of the computer, round-off errors must be considered.

Rounding errors should be considered when conversions are made between decimal digits and binary form. Input and output operations are one time when this occurs. Given the format used for REALs, the conversion REAL → decimal → REAL will yield an identity only if the REAL → decimal conversion produces a 17-decimal-digit mantissa and the calculations for the conversions are done in extra precision. This is not the case on the Series 200. Therefore, several things can be said about these conversions on the Series 200 computers:

- Up to and including 16 decimal digits are allowed when storing a number in internal form. If there are more digits, they are ignored.
- Up to and including 15 decimal digits may be output when converting a REAL for printing, display, etc. A full 16-digit conversion is not allowed because there are not 16 full digits of precision.
- It is possible for two distinct decimal numbers to map onto the same REAL number because the binary mantisa does not have enough bits to represent all 16 decimal digits. This can happen only if the decimal numbers are specified to 16-digits.
- It is possible for two distinct REAL numbers to convert to the same decimal number even if the conversion is done to 15-decimal-digit accuracy. Therefore, you cannot use a comparison of the digits in printed or displayed numbers to check for equality.
- All distinct 15 digit decimal strings have a correct distinct REAL representation, but it is not always possible to map them onto their correct representation because REAL multiplies are not done in extra precision, and the table entries are only 64 bits. In other words, the decimal → REAL conversion may produce a REAL that differs from the true representation by a maximum of two bits.

There are references at the end of this chapter to documents that contain further information on the subject of representing real numbers.

Evaluating Scalar Expressions

The Hierarchy

If you look at the expression 2+4/2+6, it can be interpreted several ways:

- 2+(4/2)+6 = 10
- (2+4)/2+6 = 9
- 2+4/(2+6) = 2.5
- (2+4)/(2+6) = .75

Computers do not deal well with ambiguity, so an arbitrary hierarchy is used for evaluating expressions to eliminate any questions about the meaning of an expression. When the computer encounters a mathematical expression, an expression evaluator is called. If you do not understand the expression evaluator, you can easily be surprised by the value returned for a given expression. In order to understand the expression evaluator, it is necessary to understand the valid elements in an expression and the evaluation hierarchy (the order of evaluation of the elements).

Six items can appear in a numeric expression; operators, constants, variables, intrinsic functions, user-defined functions and parentheses. Operators modify other elements of the expression. Constants and variables represent numeric values in the system. Functions, both intrinsic and user-defined, return a value which replaces them in the evaluation of the expression. Parentheses are used to modify the evaluation hierarchy.

The following table defines the hierarchy used by the computer in evaluating numeric expressions.

Math Hierarchy

Precedence	Operator
Highest	Parentheses; they may be used to force any order of operation
	Functions, both user-defined and machine-resident
	Exponentiation: ^
	Multiplication and division: * / MOD DIV MODULO
	Addition, subtraction, monadic plus and minus: + -
	Relational Operators: = <> < > <= >=
	NOT
	AND
Lowest	OR EXOR

When an expression is being evaluated it is read from left to right and operations are performed as encountered, unless a higher precedence operation is encountered immediately to the right of the operation encountered, or unless the hierarchy is modified by parenthesis. If the computer cannot deal immediately with the operation, it is stacked, and the evaluator continues to read until it encounters an operation it can perform. It is easier to understand if you see how an expression is actually handled. The following expression is complex enough to demonstrate most of what goes on in expression evaluation.

```
A = 5+3*(4+2)/SIN(X)+X*(1>X)+FNNeg1*(X<5 AND X>0)
```

In order to evaluate this expresion, it is necessary to have some historical data. We will assume that DEG has been executed, that X= 90, and that FNNeg1 returns -1. Evaluation proceeds as follows:

```
5+3*(4+2)/SIN(X)+X*(1>X)+FNNeg1*(X<5 AND X>0)
5+3*6/SIN(X)+X*(1>X)+FNNeg1*(X<5 AND X>0)
5+18/SIN(X)+X*(1>X)+FNNeg1*(X<5 AND X>0)
5+18/1+X*(1>X)+FNNeg1*(X<5 AND X>0)
5+18+X*(1>X)+FNNeg1*(X<5 AND X>0)
23+X*(1>X)+FNNeg1*(X<5 AND X>0)
23+X*0+FNNeg1*(X<5 AND X>0)
23+0+FNNeg1*(X<5 AND X>0)
23+FNNeg1*(X<5 AND X>0)
23+-1*(X<5 AND X>0)
23+-1*(0 AND X>0)
23+-1*(0 AND 1)
23+-1*0
23+0
23
```

The Delayed Binding Surprise

The computer delays binding of a variable to its value as long as possible. In the actual evaluation, a pointer to the location of a variable is what is stacked. This means that if a variable exists in an area of COM accessible to both the main program and a user-defined-function, is used in an expression that also calls the user-defined-function, and is modified in the function, the value of the expression can be surprising, although not unpredictable. For example, if we define a function FNNeg1 that returns a minus 1, we would expect the following lines to print 2.

```
10 COM X
20 X = 3
30 Y = X + FNNeg1
40 PRINT Y
```

However, if the user-defined-function looks like this:

```
1000  DEF FNNeg1
1010     COM X
1020     X = 500
1030     RETURN -1
1040  FN END
```

The actual result will be 499. Surprising, but not unpredictable. The same thing will happen if the variable is passed by reference and modified in the user-defined-function. This general case of occurrences is lumped together under the term side-effects, and should be avoided. Therefore, don't use a user-defined-function to modify values of variables. They are designed for returning a single value, and are best reserved for that.

Operators

There are three types of operators in BASIC, monadic, dyadic, and comparison.

- A **monadic** operator performs its operation on the expression immediately to its right.

  ```
  + - NOT
  ```

- A **dyadic** operator performs its operation on the two values it is between.

  ```
  ^ * / MOD MODULO DIV + - = <> < > <= >= AND OR EXOR
  ```

- A **comparison** operator returns a 1 (true) or a zero (false) based on the result of a relational test of the operands it separates. The comparison operators are a subset of the dyadic operators that are considered to produce *boolean* results.

  ```
  < > <= >= = <>
  ```

While the use of most operators is obvious from the descriptions in the language reference, some of the operators have uses and side-effects that are not always apparent.

Expressions, Calls, and Functions

All numeric expressions are passed by value to subprograms. Thus 5 + X is obviously passed by value. Not quite so obviously, + X is also passed by value. The monadic operator makes it an expression.

Strings in Numeric Expressions

String expressions can be directly included in numeric expressions if they are separated by comparison operators. The comparison operators always yield boolean results, and boolean results are numeric values in BASIC.

Step Functions

The comparison operators are obviously useful for conditional branching (IF...THEN statements), but are also valuable for creating numeric expressions representing step-functions. For example, let's try to represent the function:

- IF `Select` < 0
  Then `Result` = 0
- IF 0 <= `Select` < 1
  Then `Result` equals the square root of $A^2 + B^2$.
- IF `Select` >= 1 (any other value)
  Then `Result` = 15

It is possible to generate the required response through a series of `IF...THEN` statements, but it can also be done with the following expression:

```
1210 Result=(Select<0)*0+(Select>=0 AND Select<1)*SQR(A^2+B^2)+(Select>1)*15
```

While the technique may not please the purist, it actually represents the step function very well. The boolean expressions each return a 1 or 0 which is then multiplied by the accompanying expression. Expressions not matching the selection return 0, and are not included in the result. The value assigned to `Select` before the expression is evaluated determines the computation placed in the result. This technique can be used to represent other functions, but the program statement cannot exceed the maximum allowable line length.

Making Comparisons Work

If you are comparing INTEGER numbers, no special precautions are necessary. However, if you are comparing REAL values, especially those which are the results of calculations and functions, it is possible to run into problems due to rounding and other limits inherent in the system. For example, consider the use of comparison operators in IF..THEN statments to check for equality in any situation resembling the following:

```
1220   DEG
1230   A=25.3765477
1240   IF SIN(A)^2+COS(A)^2=1 THEN
1250     PRINT "Equal"
1260   ELSE
1270     PRINT "Not Equal"
1280   END IF
```

You may find that the equality test fails due to rounding errors or other errors caused by the inherent limitations of finite machines. A repeating decimal or irrational number cannot be represented exactly in any finite machine.

A good example of equality error occurs when multiplying or dividing data values. A product of two non-integer values nearly always results in more digits beyond the decimal point than exists in either of the two numbers being multiplied. Any tests for equality must consider the **exact** variable value to its greatest resolution. If you cannot guarantee that all digits beyond the required resolution are zero, there are three techniques that can be used to eliminate equality errors:

- Use the DROUND function to eliminate unwanted resolution **before** comparing results.
- Use the absolute value of the difference between the two values, and test for the difference less than a specified limit.
- Use the absolute value of the relative difference between two values, and test for the difference less than a specified limit:

```
IF ABS((C-F)/C) < 10^-Delta_power THEN PRINT "C is equal to F"
```

The following example shows the DROUND technique:

```
1050    A=32.5087
1060    B=31.625
1070      C=A*B           ! Product is 1028.08763750
1080    D=32.5122
1090    E=31.621595509
1100      F=D*E           ! Product is 1028.08763751
1110        IF C=F THEN 1130
1120          PRINT "C is not equal to F"
1130    C=DROUND(C,7)
1140    F=DROUND(F,7)
1150      IF C=F THEN
1160        PRINT "C equals F after DROUND"
1170      ELSE
1180        PRINT "C not equal to F after DROUND"
1190      END IF
1200    END
```

You can experiment with the concept by substituting other values for variables A, B, D, and E, and by changing the number of digits specified in the DROUND function.

Here is an example of the absolute value method of testing equality. In this case, a difference of less than 0.001 is assumed to be evidence of adequate equality. Using the previous example, we change technique at line 1130.

```
1130    IF ABS(C-F)<.001 THEN
1140      PRINT "C is equal to F within 0.001"
1150    ELSE
1160      PRINT "C is not equal to F within 0.001"
1170    END IF
1180    END
```

This technique has the advantage that no additional statements are invested in overhead while preparing the data for evaluation. It also enables you to easily establish tolerance limits in making value comparisons, a capability that is useful in production and testing applications.

Finally, here is an example of the relative difference method. Once again, we change the technique at line 1130.

```
1130    IF ABS((C-F)/C)< 10^-3  THEN
1140      PRINT "Relative difference between C and F less than 10^-3"
1150    ELSE
1160      PRINT "Relative difference between C and F greater than 10^-3"
1170    END IF
1180    END
```

Resident Numerical Functions

The resident functions are the functions that are part of the BASIC language (also called intrinsic). Numerous functions are included in the BASIC you are using to make mathematical modeling easier. The following functions are available:

Function	Description
ABS	Returns the absolute value of an expression.
ACS	Returns the arccosine of an expression.
ASN	Returns the arcsine of an expression.
ATN	Returns the arctangent of an expression.
BASE	Returns the lower subscript bound of a dimension of an array. (Requires MAT)
BINAND	Returns the bit-by-bit logical-and of two arguments.
BINCMP	Returns the bit-by-bit complement of two arguments.
BINEOR	Returns the bit-by-bit exclusive-or of two arguments.
BINIOR	Returns the bit-by-bit inclusive-or of two arguments.
BIT	Returns the state of a bit of the argument.
COS	Returns the cosine of the angle represented by the expression.
CRT	Returns the INTEGER 1. This is the select code of the internal CRT.
DATE	Takes a string expression and returns the number of seconds between midnight on the morning of the date represented by the string expression and 24 November -4713. (Requires CLOCK)
DET	Returns the determinant of a matrix. (Requires MAT)
DOT	Returns the inner (dot) product of two vectors. (Requires MAT)
DROUND	Rounds a number to a number of digits.
DVAL	Returns the whole number value of a binary, octal, decimal, or hexadecimal 32-bit integer. The argument is a string.
EXP	Raise the Napierian e to an power. e ≈ 2.718 281 828 459 05.
FRACT	Returns the "fractional" part of the argument.
INT	Returns the greatest integer that is less than or equal to an expression. The result is of the same type (INTEGER or REAL) as the original number.
IVAL	Returns the INTEGER value of a binary, octal, decimal, or hexadecimal 16-bit integer. The argument is a string.
KBD	Returns the INTEGER 2. This is the select code of the keyboard.
LGT	Returns the base 10 logarithm of an expression.

Function	Description
LOG	Returns the natural logarithm (Napierian base e) of an expression.
MAX	Returns the larger of a list of expressions. (Requires MAT)
MAXREAL	Returns the largest REAL number.
MIN	Returns the smaller of a list of expressions. (Requires MAT)
MINREAL	Returns the smallest REAL number.
PI	Returns the constant 3.141 592 653 589 79, an approximate value for π.
PROUND	Returns the value of the argument rounded to a power of ten.
PRT	Returns the INTEGER 701. This is the default (factory set) device selector for an external printer.
RANK	Returns the number of dimensions in an array. (Requires MAT)
RES	Returns the last live keyboard numeric result.
RND	Returns a pseudo-random number that is greater than 0 but less than 1.
ROTATE	Returns a value obtained by shifting an INTEGER representation of an argument a specific number of bit positions, with wraparound.
SC	Returns the interface select code associated with an I/O path name.
SIN	Returns the sine of the angle represented by an expression.
SIZE	Returns the number of elements in a dimension of an array. (Requires MAT)
SGN	Returns the sign of an expression: 1 if positive, 0 if 0, −1 if negative.
SHIFT	Returns a value obtained by shifting an INTEGER representation of an argument a specific number of bit positions, without wraparound.
SQR	Returns the square root of an expression.
SUM	Returns the sum of all the elements in an array. (Requires MAT)
TAN	Returns the tangent of the angle represented by an expression.
TIME	Returns the number of seconds between midnight and the time represented by the string argument. (Requires CLOCK)

Dealing with Angles and Such

Six functions are provided for dealing with angles and angular measure (SIN, ASN, COS, ACS, TAN, ATN). The default mode for all angular measure is radians. Degrees can be selected with the DEG statement. Radians may be re-selected by the RAD statement. It is a good idea to explicitly set a mode for any angular calculations, even if you are using the default (radian) mode. This is especially important in writing subprograms, as the subprogram inherits the angular mode from the context that calls it. When the calling context is restored, the angle mode is also restored.

Range Limits

It is sometimes necessary to limit the range of excursion of a variable (as in the discussion of REAL to INTEGER conversions mentioned in the introduction to this chapter). While it is possible to do this with IF...THEN statements:

```
200 IF X>Maxx THEN X = Maxx
210 IF X<Minx THEN X = Minx
```

it is more convenient to use the MAX and MIN functions.

```
200  X = MIN(MAX(X,Minx),Maxx)
```

Note that MAX is used to establish the lower bound, and MIN is used to establish the upper bound. If you think about it a minute, it makes sense.

Rounding

Rounding occurs frequently in computer operations. The most common rounding occurs in printouts and displays, where it can be handled effectively with a USING clause in the output operation. For details see the section on Formatted Output in the Using a Printer chapter. This works in statements such as PRINT, LABEL, OUTPUT and DISP.

Sometimes it is necessary to round a number in a calculation, to eliminate unwanted resolution. There are two basic types of rounding; rounding to a total number of decimal digits and rounding to a number of decimal places (limiting fractional information) . Both types of rounding have their own application in programming.

There is a tendency for the number of decimal places to grow as calculations are performed on the results of other calculations. One of the first things covered in training for engineering and the sciences is how to handle the growth of the number of decimal places in a calculation. If the initial measurements from an experiment produced three digits of information per reading, it is very misleading to produce a seven-digit number as the result of a long series of calculations. The DROUND function allows you to eliminate the unwanted digits, to produce more realistic calculations and answers.

```
X=DROUND(SQR(Y^3+Gr^3),3)
```

It is also possible to round to a number of decimal places. The idea is to eliminate decimal representation beyond a specific power of ten. The PROUND function allows you to perform a round to any specified power of ten.

```
200   X=PROUND(X1,Places)
```

Random Numbers

The RND function returns a psuedo-random random number between 0 and 1. Since many modeling systems require random numbers with arbitrary ranges, it is necessary to scale the numbers.

```
200    R=INT(RND*Range)+Offset
```

The above statement will return an integer between `Offset` and `Offset + Range`.

The random number generator is seeded with the value 37 480 660 at power-on, SCRATCH, SCRATCH A, and prerun. The pattern period is $2^{31} - 2$. You can change the seed with the RANDOMIZE statement, which will give a new pattern of numbers.

Binary Operations

In actuality, all operations the computer performs are binary. There are, however, some logical and bit-level operations available on the computer that are commonly called binary operations. When any of these operations are used, the arguments are first converted to INTEGER (if they are not already in the correct form) and then the specified operation is performed. It is best to restrict bit-oriented binary operations to declared INTEGERs. If it is necessary to operate on a REAL, make sure the precautions described under Conversions, at the beginning of this chapter, are employed, to avoid INTEGER overflow.

Array Operations

An array is a multi-dimensioned structure of variables that are given a common name. The array can have one through six dimensions. Each location in an array can contain one variable value, and each value has the characteristics of a single variable, depending on whether the array consists of REAL or INTEGER values (string arrays are discussed in Chapter 5, "String Manipulation"). Note that many of the statements that deal with arrays (such as MAT) require the MAT BIN.

A one-dimensional array consists of *n* elements, each identified by a single subscript. A two-dimensional array consists of *m* times *n* elements where *m* and *n* are the maximum number of elements in the two respective dimensions. Arrays require a subscript in each dimension, in order to locate a given element of the array. Up to six dimensions can be specified for any array in a program. REAL arrays require eight bytes of memory for each element, plus overhead; so you can see that large arrays can demand massive memory resources.

An undeclared array is given as many dimensions as it has subscripts in its lowest-numbered occurrence. Each dimension of an undeclared array has an upper bound of ten. Space for these elements is reserved whether you use them or not.

Dimensioning an Array

Before you use an array, you should tell the system how much memory to reserve for it. This is called "dimensioning" an array. You can dimension arrays with the DIM, COM, ALLOCATE, INTEGER, or REAL statements. An array is a type of variable and as such follows all rules for variable names. Unless you explicitly specify INTEGER type in the dimensioning statement, arrays default to REAL type. The same array can only be dimensioned once in a context[1]. However, as we explain later in this section, arrays can be REDIMensioned.

When you dimension an array, the system reserves space in internal memory for it. The system also sets up a table which it uses to locate each element in the array. The location of each element is designated by a unique combination of subscripts, one subscript for each dimension. For a four-dimensional array, for instance, each element is identified by four subscript values. Each unique set of subscript values points to one, and only one, array element.

The actual size of an array is governed by the number of dimensions and the subscript range of each dimension. If **A** is a three-dimensional array with a subscript range of 1 thru 4 for each dimension, then its size is 4 x 4 x 4, 64 elements.

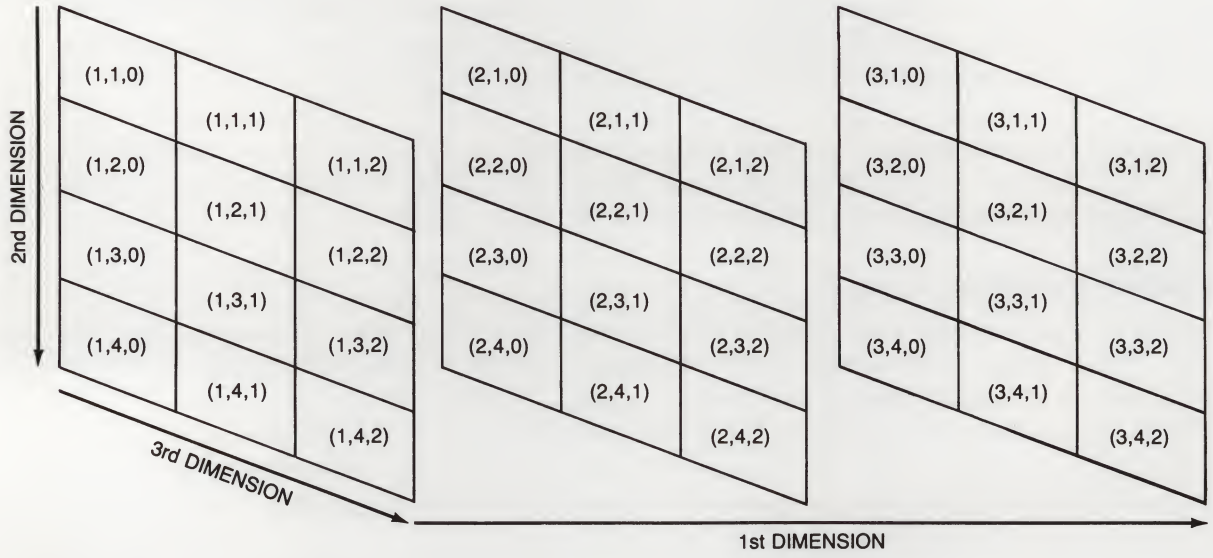
When you dimension an array, therefore, you must give not only the number of dimensions but also the subscript range of each dimension. Subscript ranges can be specified by giving the lower and upper bounds, or by giving just the upper bound. If you give only the upper bound, the lower bound defaults to the current option base setting.

1 There is one exception to this rule: If you ALLOCATE an array, and then DEALLOCATE it, you can dimension the array again.

Each context initializes to an option base of 0 (but arrays appearing in COM statements with an (*) will keep the base with which they were originally dimensioned). However, you can set the option base to 1 with the OPTION BASE command. You can have only one OPTION BASE statement in a context, and it must precede all explicit variable declarations.

The following examples illustrate some of the flexibility you have in dimensioning arrays.

```
10  OPTION BASE 1
20  DIM A(3,4,0:2)
```



	Size	Lower Bound	Upper Bound
1st Dimension	3	1	3
2nd Dimension	4	1	4
3rd Dimension	3	0	2

In this example we portray the first dimension as planes, the second dimension as rows, and the third dimension as columns. In general, the last two dimensions of any array always refer to rows and columns, respectively. When we discuss two-dimensional arrays, the first dimension will always represent rows, and the second dimension will always represent columns. Note also in the above example that the first two dimensions use the default setting of 1 for the lower bound, while the third dimension explicitly defines 0 as the lower bound. The numbers in parentheses are the subscript values for the particular elements. These are the numbers you use to identify each array element.

```
10  OPTION BASE 1
20  COM B(1:5,2:6)
        .
        .
```

(1,2)	(1,3)	(1,4)	(1,5)	(1,6)
(2,2)	(2,3)	(2,4)	(2,5)	(2,6)
(3,2)	(3,3)	(3,4)	(3,5)	(3,6)
(4,2)	(4,3)	(4,4)	(4,5)	(4,6)
(5,2)	(5,3)	(5,4)	(5,5)	(5,6)

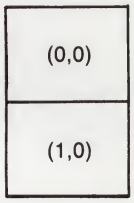
	Size	Lower Bound	Upper Bound
1st Dimension	5	1	5
2nd Dimension	5	2	6

```
10   OPTION BASE 1
20   ALLOCATE INTEGER C(2:4,-2:2)
        .
        .
```

(2,−2)	(2,−1)	(3,0)	(2,1)	(2,2)
(3,−2)	(3,−1)	(3,0)	(3,1)	(3,2)
(4,−2)	(4,−1)	(4,0)	(4,1)	(4,2)

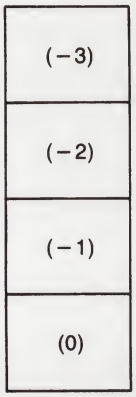
	Size	Lower Bound	Upper Bound
1st Dimension	3	2	4
2nd Dimension	5	−2	2

```
10    OPTION BASE 0
20    REAL D(1,0)
            .
            .
```



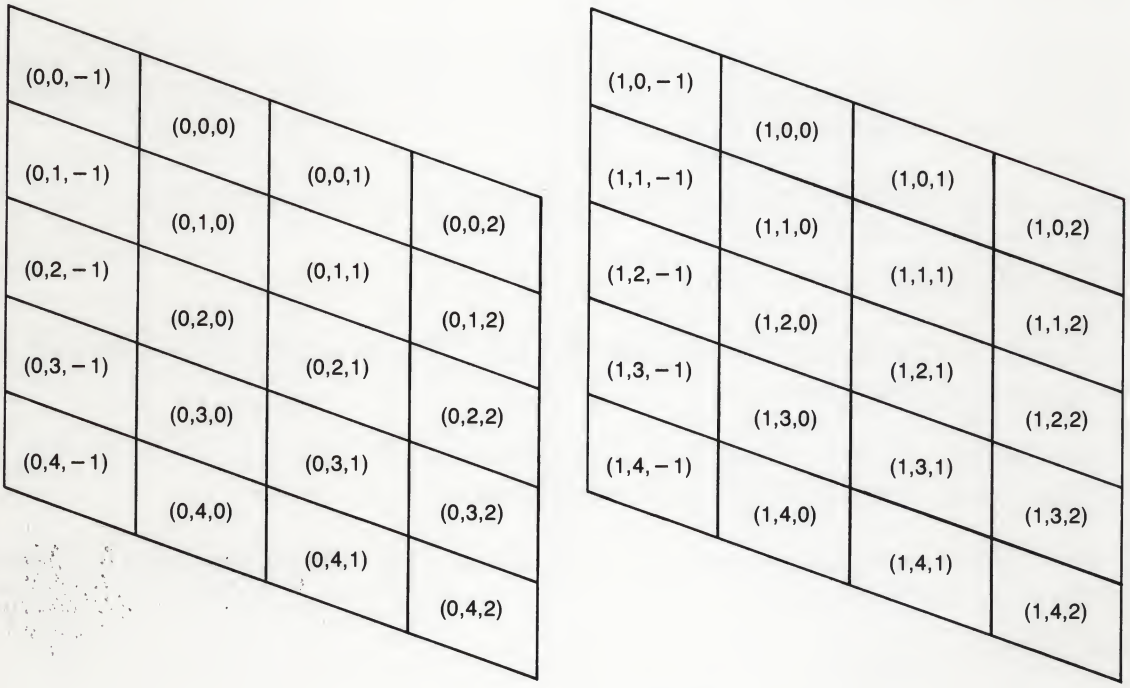
	Size	Lower Bound	Upper Bound
1st Dimension	2	0	1
2nd Dimension	1	0	0

```
10    COM E(-3:0)
            .
            .
```



	Size	Lower Bound	Upper Bound
1st Dimension	4	−3	0

```
10 OPTION BASE 0
20 INTEGER F(1,4,-1:2)
          .
          .
```



	Size	Lower Bound	Upper Bound
1st Dimension	2	0	1
2nd Dimension	5	0	4
3rd Dimension	4	−1	2

Arrays are limited to six dimensions, and the subscript range for each dimension must lie between −32767 and 32767. (REDIM and ALLOCATE allow the subscript range to go down to −32768, but the total size of each dimension must be less than 32768 elements.) For the most part, we use only two-dimensional examples since they are easier to illustrate. However, the same principles apply to arrays of more than two dimensions as well.

Note

Throughout this chapter we will be using DIM statements without specifying what the current option base setting is. Unless explicitly specified otherwise, all examples in this chapter use option base 1.

As an example of a four-dimensional array, consider a five-story library. On each floor there are 20 stacks, each stack contains 10 shelves, and each shelf holds 100 books. To specify the location of a particular book you would give the number of the floor, the stack, the shelf, and the particular book on that shelf. We could dimension an array for the library with the statement:

```
DIM Library(5,20,10,100)
```

This means that there are 100,000 book locations. To identify a particular book you would specify its subscripts. For instance, `Library(2,12,3,35)` would identify the 35th book on the 3rd shelf of the 12th stack on the 2nd floor.

We can imagine accessing a particular page of a book by using a 5-dimensional array. For instance, if we dimension an array,

```
DIM Page(5,20,10,100,200)
```

then `Page(1,7,2,19,130)` would designate page 130 of the 19th book on the 2nd shelf of the 7th stack on the 1st floor.

We could specify words on pages by using a 6-dimensional array. Six dimensions is the maximum, though, so we could not specify letters of words.

Also, you can dimension more than one array in a single statement by separating the declarations with a comma. For instance,

```
10    DIM A(1,3,4),B(-2:0,2:5),C(5)
```

would dimension all three arrays: `A`, `B`, and `C`.

Problems with Implicit Dimensioning

In any environment, an array must have a dimensioned size. This size can be passed into an environment through a passed parameter list or a COM statement. It may be explicitly dimensioned through COM, INTEGER, REAL or ALLOCATE. It can also be implicitly dimensioned through a subscripted reference to it in a program statement **other than** a MAT or a REDIM statement. An attempt to use an array that does not have a dimensioned size in the current environment in a MAT or REDIM statement will result in an error. In other words, MAT and REDIM statements cannot be used to implicitly dimension an array.

Finding Out the Dimensions of an Array

There are a number of statements that allow you to determine the size and shape of an array. To find out how many dimensions are in an array, use the RANK function. For instance:

```
OPTION BASE 0
DIM F(1,4,-1:2)
PRINT RANK (F)
```

would print 3.

The SIZE function returns the size (number of elements) of a particular dimension. For instance,

```
SIZE (F,2)
```

would return 5, the number of elements in F's second dimension.

To find out what the lower bound of a dimension is, use the BASE statement. Referring again to array F

```
BASE (F,1)
```

would return a 0, while,

```
BASE (F,3)
```

would return a -1.

By using the SIZE and BASE statements together, you can determine the upper bounds of any dimension (e.g., SIZE + BASE $-$ 1 = Upper Bound).

It may seem pointless to have all these functions that return the dimension specifications which you yourself assigned. After all, if you assigned the dimensions, you should know what they are; and if you forget, you can always look at the appropriate dimensioning statement. However, these functions are powerful tools for writing programs that perform functions on an array regardless of the array's size or shape. In addition, the system automatically redimensions arrays during certain operations. The functions discussed above provide you with a means for determining the new dimensions. As an example of a general purpose program utilizing these statements, consider the subprogram below which prints a two-dimensional array in rows and columns.

```
100  SUB Printmat(Array(*))
110  OPTION BASE 1
120  FOR Row=BASE(Array,1) TO SIZE(Array,1)+BASE(Array,1)-1
130    FOR Column=BASE(Array,2) TO SIZE(Array,2)+BASE(Array,2)-1
140      PRINT USING "DDDD.DD,XX,#";Array(Row,Column)
150    NEXT Column
160    PRINT
170  NEXT Row
180  SUBEND
```

Filling an Array

Once an array has been dimensioned, the next step is to fill it with useful values. Initially, every element in an array equals zero. There are a number of different ways to change these values. The most obvious is to assign a particular value to each element. This is done by specifying the element's subscripts. For instance, the statement,

```
A(3,4)=13
```

would assign the value 13 to the element in the third row and fourth column of A. You must give enough subscripts for the system to identify a single element. For a three-dimensional array, for instance, you would provide three subscripts. All subscripts, moreover, must lie within the dimensioned range. If you use out-of-range subscripts, the system returns an error.

For some applications, you may want to initialize every element in an array to some particular value. You can do this by assigning a value to the array name. However, you must precede the assignment with the MAT keyword. For example,

```
MAT A= (10)
```

will assign the value 10 to every element in array A regardless of A's size. Note that the numeric expression on the right-hand side of the assignment must be enclosed in parentheses.

Another way to assign values to an array is by using the READ and DATA statements. The DATA statement allows you to create a stream of data items, and the READ statement enables you to enter the data stream into an array. For example:

```
10  OPTION BASE 1
20  DIM A(3,3)
30  DATA -4,36,2.3,5,89,17,-6,-12,42
40  READ A(*)
50  END
```

The asterisk in line 40 is used to designate the entire array rather than a single element. Note also that the right-most subscript varies fastest. In this case, it means that the system fill an entire row before going to the next one. The READ/DATA statements are discussed further in Chapter 7.

If we use the Printmat subprogram (shown on previous page) to display A, we get:

```
-4.00      36.00      2.30
 5.00      89.00     17.00
-6.00     -12.00     42.00
```

Copying Arrays

A fourth way to fill an array is to copy the elements from one array into another. Suppose, for example, that you have the two arrays A and B shown below.

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 3 & 5 \\ 8 & 2 \\ 1 & 7 \end{bmatrix}$$

Note that A is a 3x3 array which is filled entirely with 0's, while B is a 3x2 array filled with non-zero values. To copy B to A, we would execute:

```
MAT A=B
```

Again, you must precede the assignment with MAT. The system will automatically redimension the "result array" (the one on the left-hand side of the assignment) so that it is the same size as the "operand array" (the one on the right side of the equation.) There are two restrictions on redimensioning an array.

- The two arrays must have the same rank (e.g., the same number of dimensions.)
- The dimensioned size of the result array must be at least as large as the current size of the operand array.

If the system cannot redimension the result array to the proper size, it returns an error.

Automatic redimensioning of an array will not affect the lower bounds, only the upper bounds. So the BASE values of each dimension of the result array will remain the same. Also keep in mind that the size restriction applies to the *dimensioned* size of the result array and the *current* size of the operand array. Suppose we dimension arrays A, B and C to the following sizes:

```
10   OPTION BASE 1
20   DIM A(3,3),B(2,2),C(2,4)
                  .
                  .
```

We can execute,

```
MAT A=B
```

since A is dimensioned to 9 elements and B is only 4 elements. The copy automatically redimensions A to a 2x2 array. Nevertheless, we can still execute:

```
MAT A=C
```

This is because the nine elements originally reserved for A remain available until the program is scratched. A now becomes a 2x4 matrix. After `MAT A=C`, we could not execute:

```
MAT B=A
  or
MAT B=C
```

since in each of these cases, we are trying to copy a larger array into a smaller one. But we could execute

```
MAT C=A
```

after the original `MAT A=B` assignment, since C's dimensioned size (8) is larger than A's current size (4).

Extracting Values From Arrays

As with entering values into arrays, there are a number of ways to extract values as well. To extract the value of a particular element, simply specify the element's subscripts. For instance, the statement,

```
X=A(3,4,2)
```

would assign the value of the element occupying the given location in A to the variable X. The system will automatically convert variable types. For example, if you assign an element from a Real array to an Integer variable, the system will perform the necessary rounding.

Certain operations (e.g., PRINT, OUTPUT, ENTER and READ) allow you to access all elements of an array merely by using an asterisk in place of the subscript list. The statement,

```
PRINT A(*);
```

would display every element of A on the current PRINTER IS device. The elements are displayed in order, with the rightmost subscripts varying fastest. The semi-colon at the end of the statement is equivalent to putting a semi-colon between each element. When they are displayed, therefore, they will be separated by a space. (The default is to place elements in successive columns.)

The asterisk is also used to pass an array as a parameter to a function or subprogram. For instance, to pass an array A to the Printmat subprogram listed earlier, we would write:

```
Printmat (A(*))
```

In addition to extracting the values of elements in an array, there are also functions that compute the sum of an entire row or column of an array. However, since these functions are limited to two-dimensional arrays, we will reserve their discussion for the section on matrices. The statement that returns the sum of *all* elements in an array, however, works for arrays of any dimension. Given the array A below,

$$A = \begin{bmatrix} 4 & 2 & -1 \\ 3 & 8 & 16 \\ -5 & 2 & 0 \end{bmatrix}$$

the function,

```
SUM(A)
```

would return 29.

Redimensioning Arrays

In our discussion of copying arrays we saw that the system automatically redimensions an array if necessary. You can also explicitly redimension an array with the REDIM statement. As with automatic redimensioning, the following two rules apply to all REDIM statements:

- A REDIMed array must maintain the same number of dimensions.
- You cannot REDIM an array so that it contains more elements than it was originally dimensioned to hold.

Suppose A is the 3x3 array shown below.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

To redimension it to a 2x4 array, you would execute:

```
REDIM A(2,4)
```

The new array now looks like the figure below:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

Note that it retains the values of the elements, though not necessarily in the same locations. For instance, A(2,1) in the original array was 4, whereas in the redimensioned array it equals 5. If we REDIMed A again, this time to a 2x2 array, we would get:

```
REDIM A(0:1,0:1)
```

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

We could then initialize all elements to 0:

```
MAT A= (0)
```

$$A = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

It is also important to realize that elements that are out of range in the REDIMed array still retain their values. The fifth thru ninth elements in A still equal 5 thru 9 even though they are now inaccessible. If we REDIM A back to a 3x3 array, these values will reappear. For example:

```
REDIM A(3,3)
```

produces:

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

One of the major strengths of the REDIM statement is that it allows you to use variables for the subscript ranges: this is not allowed when you originally dimension an array. In effect, this enables you to dynamically dimension arrays. This should not be confused with the ALLOCATE statement which allows you to dynamically reserve memory for arrays. In the example below, for instance, we enter the dimensions from the keyboard.

```
10   OPTION BASE 1
20   DIM A(100,100)   ! 1000 ELEMENT ARRAY
30   INPUT "Enter lower and upper bounds of dimensions",
     Low1,Up1,Low2,Up2
40   IF (Up1-Low1+1)*(Up2-Low2+1)>10000 THEN Too_big
50   REDIM A(Low1:Up1,Low2:Up2)
```

Line 40 tests to see whether the new dimensions are too big. If so, program control is passed to a line labelled "Too_big". If line 40 were not present, the REDIM statement would return an error if the dimensions were too large.

Arrays and Arithmetic Operators

It is possible to multiply, divide, add, and subtract scalars to an array, as well as to add, subtract, multiply, and divide one array to another. All arithmetic functions involving arrays must be preceded by the MAT keyword. The specified operation is performed on each individual element in the operand array(s) and the results are placed in the result array. The result array must be dimensioned to be at least as large as the current size of the operand array(s). If it is of a different shape than the operand array(s), the system will redimension it. Given the array A below, note how these arithmetic functions are performed.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
MAT B= A+(3)
```

$$B = \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

```
MAT C= B/(2)
```

C

$$\begin{bmatrix} 2 & 2.5 & 3 \\ 3.5 & 4 & 4.5 \\ 5 & 5.5 & 6 \end{bmatrix}$$

```
MAT C= C*(1+1+1)
```

C

$$\begin{bmatrix} 6 & 7.5 & 9 \\ 10.5 & 12 & 13.5 \\ 15 & 16.5 & 18 \end{bmatrix}$$

Note that the result array can be the same as the operand array. Also, the scalar must be enclosed in parentheses.

In addition to performing arithmetic operations with scalars, you can also add, subtract, divide and multiply two arrays together. Except for multiplication with an asterisk, which is described later, these functions proceed as follows: Corresponding elements of each operand array are processed according to the specified operation, and the result is placed in the result array. The two operand arrays must be exactly the same size though their particular subscript ranges can be different. For multiplication, use a period rather than an asterisk. Using arrays A and B above, the statement,

```
MAT D= A+B
```

would give the array:

D

$$\begin{bmatrix} 5 & 7 & 9 \\ 11 & 13 & 15 \\ 17 & 19 & 21 \end{bmatrix}$$

The statement,

```
MAT B= A.B
```

would give:

B

$$\begin{bmatrix} 4 & 10 & 18 \\ 28 & 40 & 54 \\ 70 & 88 & 108 \end{bmatrix}$$

Again, the dimensioned size of the result array must be as large as the current size of each operand array. The two operand arrays must be identical in shape and size, but not necessarily in subscript ranges. For instance, A and B could have been dimensioned:

```
10    DIM A(1:3,2:4),B(-1:1,0:2)
```

Boolean Arrays

In addition to the arithmetic operators, you can also use relational operators with arrays. The result is a boolean[1] array (e.g., an array composed entirely of 1's and 0's). Given array B above, suppose you wanted to know how many elements were greater than 50. First you execute the statement,

```
MAT F= B>(50)
```

which results in the array:

$$F = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Then you execute the statement,

```
PRINT SUM(F)
```

which causes the computer to display "4" on the current PRINTER IS device.

You can also compare two arrays to each other. If, for example, you wanted to compare the two arrays below,

$$A = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 8 & 7 \\ 1 & 4 & 6 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 3 & 4 \\ 2 & 0 & 7 \\ 1 & 4 & 4 \end{bmatrix}$$

you could execute the statement:

```
MAT C= A=B
```

By looking at C, you can tell which elements are the same for both A and B.

$$C = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

1 Strictly speaking, these are not really boolean arrays since the values of the elements are not TRUE and FALSE.

Reordering Arrays

The MAT REORDER statement allows you to re-arrange an array so that one dimension is in a particular order. The new order is specified in a vector (a vector is a one-dimensional array). The vector contains the subscripts of the reordered dimension in their new order. The subscripts must correspond to the array's current dimensions and subscript ranges. Suppose A is the array below. Let us also assume that A has been dimensioned in OPTION BASE 1, and that the upper bound to both dimensions is 3.

$$\begin{matrix} \textbf{A} \\ \begin{bmatrix} 1 & 3 & 2 \\ 4 & 5 & 7 \\ 6 & 8 & 9 \end{bmatrix} \end{matrix}$$

To reverse the order of the rows, we would first dimension a vector,

```
10 DIM Reverse(3)
```

and then assign its elements the following values:

```
Reverse(1)=3
Reverse(2)=2
Reverse(3)=1
```

The array `Reverse` now appears:

$$\begin{matrix} \textbf{Reverse} \\ \begin{bmatrix} 3 & 2 & 1 \end{bmatrix} \end{matrix}$$

If we execute the statement,

```
MAT REORDER A BY Reverse
```

the result array will be:

$$\begin{matrix} \textbf{A} \\ \begin{bmatrix} 6 & 8 & 9 \\ 4 & 5 & 7 \\ 1 & 3 & 2 \end{bmatrix} \end{matrix}$$

Note that the rows are exchanged, rather than the columns. This is because the default is to re-order the 1st dimension. However, you can override the default by specifying a particular dimension to be re-ordered. For example, if we wanted to reverse columns rather than rows, we could use the same vector, but this time specify dimension 2:

```
MAT REORDER A BY Reverse,2
```

The transformation would be:

$$\begin{matrix} \textbf{A} \\ \begin{bmatrix} 6 & 8 & 9 \\ 4 & 5 & 7 \\ 1 & 3 & 2 \end{bmatrix} \end{matrix} \quad \Rightarrow \quad \begin{matrix} \textbf{A} \\ \begin{bmatrix} 9 & 8 & 6 \\ 7 & 5 & 4 \\ 2 & 3 & 1 \end{bmatrix} \end{matrix}$$

Remember that although our examples are confined to two dimensions for illustrative purposes, the same principles apply to arrays of three and more dimensions. In a three-dimensional array, for instance, reordering the 1st dimension would reorder planes rather than rows or columns.

In most cases, rather than creating a reorder vector and assigning values to it, you will already have a vector as the result of a sort operation. This is described in the next section.

Sorting Arrays

A frequent operation performed on arrays is a sort. Sorting an array rearranges the array so that one dimension (which you specify) is in numerical order. Given the array A below, watch how the MAT SORT changes it.

A

5	6	8
3	5	1
2	4	8

```
MAT SORT A(*,1)
```

A

2	4	8
3	5	1
5	6	8

The asterisk specifies the dimension to be sorted, and the subscript(s) tells which elements in that dimension to use as the sorting values. In the example above, we told the system to sort rows (asterisk is located in the first subscript position), and to use the first element in each row as the sorting value. With the new array A (from the sort performed above), the following statement will sort columns using the second element in each column as the sorting value.

```
MAT SORT A(2,*)
```

A

8	2	4
1	3	5
8	5	6

The key values in this sort are 1, 3 and 5, the second elements in each column. Sorting by placing the lowest values first is known as sorting by "ascending" order. This is the default. You can also sort by "descending" order by specifying the secondary keyword DES. For instance, the statement,

```
MAT SORT A(*,2) DES
```

would produce the following transformation:

A

8	2	4
1	3	5
8	5	6

$\Rightarrow$

A

8	5	6
1	3	5
8	2	4

Sometimes the values of two or more sorting elements are the same. For instance, if we sorted A by rows using the first element,

```
MAT SORT A(*,1)
```

we get:

$$\begin{matrix}\text{A}\\ \begin{bmatrix}8&5&6\\1&3&5\\8&2&4\end{bmatrix}\end{matrix} \Rightarrow \begin{matrix}\text{A}\\ \begin{bmatrix}1&3&5\\8&5&6\\8&2&4\end{bmatrix}\end{matrix}$$

The first elements in the last two rows are the same, so the system leaves them in the order they held before the sort. However, you can specify a second sort element to be used in the case of ties. We could execute:

```
MAT SORT A(*,1),(*,2)
```

This tells the system to sort by rows using the first element as the sorting value; and in the case of ties, to use the second element. The result array would be:

$$\begin{matrix}\text{A}\\ \begin{bmatrix}1&3&5\\8&5&6\\8&2&4\end{bmatrix}\end{matrix} \Rightarrow \begin{matrix}\text{A}\\ \begin{bmatrix}1&3&5\\8&2&4\\8&5&6\end{bmatrix}\end{matrix}$$

The maximum number of sorting keys is 25.

If a key is specified that is recognized by the system as rendering all other keys redundant (such as a non-substringed key for a one dimensional string array) no other keys can be specified. However, if the computer cannot tell that keys are redundant (such as `MAT SORT A(*,X),A(*,Y)` with `X` equal to `Y`) it will permit redundant keys. Redundant keys will slow down execution of the MAT SORT statement. If you include the DES secondary word, it refers only to the sort element which immediately precedes it.

Sorting to a Vector

So far, all of our sort examples have actually re-arranged the array in question. Alternatively, you can record the new order in a vector and leave the array intact. The vector must have been dimensioned to have at least as many elements as the current size of the array being sorted. If necessary, the system will redimension the vector. Thus, executing the statement:

```
MAT SORT A(3,*) TO Vect
```

with the array A:

$$\begin{matrix}\text{A}\\ \begin{bmatrix}1&3&5\\8&2&4\\8&5&6\end{bmatrix}\end{matrix}$$

The array A remains unchanged, but the vector `Vect` now contains the values:

$$\begin{matrix}\text{Vect}\\ \begin{bmatrix}2&3&1\end{bmatrix}\end{matrix}$$

This assumes that the array A has been dimensioned so that the subscript range is 1 thru 3. If A had been dimensioned.

```
10  DIM A(1:3,-1:1)
```

then Vect would contain the values:

Vect

$$\begin{bmatrix} 0 & 1 & -1 \end{bmatrix}$$

This vector should look very reminiscent of the vectors used to reorder arrays. And, in fact, you can use these vectors in a MAT REORDER statement to rearrange the array. That is, we could now execute:

```
MAT REORDER A BY Vect,2
```

and the new array would be:

A

$$\begin{bmatrix} 3 & 5 & 1 \\ 2 & 4 & 8 \\ 5 & 6 & 8 \end{bmatrix}$$

Note that the dimension number in the MAT REORDER statement corresponds to the position of the asterisk in the MAT SORT statement.

Sorting to a vector is particularly useful if you want to sort the same array along different dimensions or using different sort elements. Each sort can be stored in a vector to be used later. Meanwhile, the original array remains unchanged.

In addition, sorting to a vector allows you to use the same sorting order with parallel arrays. That is, if you have several arrays that contain data about the same elements, you can sort one of them, and then use that same sorting order to reorder the others.

Finally, sorting to a vector enables you to manipulate an unsorted array as if it were sorted. For instance, suppose you have the array shown below:

A

$$\begin{bmatrix} 2 & 7 & 4 \\ 0 & 1 & 8 \\ 5 & 3 & 1 \end{bmatrix}$$

Let us also assume that the subscript range for each dimension in A is 1 thru 3. If we sort A to a vector B,

```
MAT SORT A(*,1) TO B
```

we can then use B to define elements in A. For instance, to get the value of A(1,1) in its sorted form, we could write:

```
X=A(B(1),1)
```

In this case, X would equal 0. By incrementing the subscript value of B, we can simulate a sorted A.

We should point out again that although these examples are two-dimensional, the same principles apply to arrays of any rank. You must have one, and only one, asterisk in the subscript list of a sort. The other subscripts specify the particular elements to be used as the sorting keys.

Matrices and Vectors

A two-dimensional numeric array is called a "matrix" and a one-dimensional numeric array is called a "vector". An entire branch of mathematics is devoted to matrices and vectors, and their applications are surprisingly broad. There are certain operations that are frequently performed on matrices which have been incorporated in BASIC. Keep in mind that the functions described in this section apply only to two-dimensional, and occasionally one-dimensional arrays, but never to arrays of more than two dimensions.

Matrix Multiplication

You may recall from our discussion of arrays and arithmetic operations that the asterisk (*) is reserved for matrix multiplication. If A is an *i-by-k* matrix and B is a *k-by-j* matrix, then C=A*B is defined by the following equation:

$$C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}$$

Translated into english, this equation means that the element in the *ith* row and *jth* column of the product (C) is the sum of the products by pairs of the elements in the *ith* row of A and the *jth* column of B. A couple of examples will help make this clear.

Suppose A and B are the matrices shown below.

$$\mathbf{A} = \begin{bmatrix} 3 & 8 & 2 \\ 1 & 6 & 5 \\ 4 & 2 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 1 & -2 & 3 \\ -6 & 4 & 7 \\ 0 & 8 & 2 \end{bmatrix}$$

If C=A*B, then :

```
C(1,1)=A(1,1)*B(1,1)+A(1,2)*B(2,1)+A(1,3)*B(3,1)=(3*1)+(8*-6)+(2*0)=-45
C(2,1)=A(2,1)*B(1,1)+A(2,2)*B(2,1)+A(2,3)*B(3,1)=(1*1)+(6*-6)+(5*0)=-35
C(3,2)=A(3,1)*B(1,2)+A(3,2)*B(2,2)+A(3,3)*B(3,2)=(4*-1)+(2*4)+(0*8)=0
```

Following this procedure for each element in C, we get the matrix shown below.

```
MAT C=A*B
```

$$\mathbf{C} = \begin{bmatrix} -45 & 42 & 69 \\ -35 & 62 & 55 \\ -8 & 0 & 26 \end{bmatrix}$$

Note that the product is a 3x3 matrix. There are three general rules to matrix multiplication:

- Multiplication between two matrices is legal only if the second dimension of the first array is the same size as the first dimension of the second array. That is, the two inner dimensions must be the same.
- The result matrix will have the same number of rows as the first operand matrix and the same number of columns as the second operand matrix. That is, the dimensions of the result matrix will be the same as the outer dimensions of the operand matrices.
- The result array cannot be the same as either of the operand arrays. For example,

```
MAT A= A*B
```

  is an illegal statement.

If A is a 2x3 matrix and B is a 3x2 matrix, A*B will result in a 2x2 matrix. B*A, on the other hand, produces a 3x3 matrix. Given the two matrices below, you can see how their position in the equation affects the product.

$$\mathbf{A}\begin{bmatrix} 6 & 8 & -1 \\ 2 & -3 & 4 \end{bmatrix} \qquad \mathbf{B}\begin{bmatrix} -1 & 1 \\ 2 & -2 \\ 3 & 4 \end{bmatrix}$$

$$\mathbf{A*B}\begin{bmatrix} 7 & -14 \\ 4 & 24 \end{bmatrix} \qquad \mathbf{B*A}\begin{bmatrix} -4 & -11 & 5 \\ 8 & 22 & -10 \\ 26 & 12 & 13 \end{bmatrix}$$

Multiplication With Vectors

We described a vector as a one-dimensional array. For instance,

```
10 DIM A(3)
```

would create a vector with three elements and a rank of 1. Suppose we give A the values shown below.

$$\mathbf{A}\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

Notice that we have portrayed A as a row vector. We could have just as easily portrayed A as a column vector:

$$\mathbf{A}\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

So which is it? A row vector or a column vector? Actually, a vector can behave like either depending on its position in an equation. If a vector is the first operand in a multiplication, then it acts like an 1Xn array (row vector); if it's the second operand, it behaves like a nX1 array (column vector); and if it's the result array, it can act like either. A few examples will help illustrate these principles. Let A be the vector shown above, and B, C, and D be the arrays shown below.

$$\mathbf{B}\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix} \qquad \mathbf{C}\begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix} \qquad \mathbf{D}\begin{bmatrix}2\\2\\2\end{bmatrix}$$

Let us suppose that D has been explicitly defined as a two-dimensional array:

```
10  DIM D(3,1)
```

If we execute:

```
MAT C= A*D
```

we get:

$$\mathbf{C}\begin{bmatrix}12\end{bmatrix}$$

Since A is the first operand, it behaves like a 1x3 matrix. The equation, therefore, is:

$$\mathbf{C}=\begin{bmatrix}1&2&3\end{bmatrix}*\begin{bmatrix}2\\2\\2\end{bmatrix}$$

The result is a 1x1 matrix. If we try to reverse the order:

```
MAT C=D*A
```

the system returns:

```
ERROR 16  Improper dimensions
```

This is because we tried to multiply a 3x1 matrix by a 3x1 matrix:

$$\mathbf{C}=\begin{bmatrix}2\\2\\2\end{bmatrix}*\begin{bmatrix}1\\2\\3\end{bmatrix}$$

Since the inner dimensions are not the same, the system returns an error. Suppose we try:

```
MAT C=B*A
```

In this case, we are multiplying a 3x3 array to a column vector.

$$C = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

The result is a 3x1 matrix:

$$\begin{matrix} \mathbf{C} \\ \begin{bmatrix} 14 \\ 32 \\ 50 \end{bmatrix} \end{matrix}$$

If the result array is a vector, then it will behave like either a row vector or a column vector depending on which is called for. The only other possibility is if both operand arrays are vectors. In this case, the result is always a 1x1 array. For instance, if A and B are the vectors below,

$$\begin{matrix} \mathbf{A} \\ \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} \end{matrix} \qquad \begin{matrix} \mathbf{B} \\ \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix} \end{matrix}$$

then multiplying A by B results in the equation:

```
MAT C=A*B
```

$$C = \begin{bmatrix} 2 & 4 & 6 \end{bmatrix} * \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix}$$

C equals −2. Reversing the operand arrays, we get:

```
MAT C=B*A
```

$$C = \begin{bmatrix} 0 & 1 & -1 \end{bmatrix} * \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}$$

Again, C equals −2. Because the product of two vectors is always a single element, BASIC has a DOT function that multiplies two vectors and comes up with a Real or Integer numeric. For example,

```
X=DOT(A,B)
```

would assign the value −2 to X. If both vectors are Integer, then the product is an Integer. Otherwise, the product is Real. The two vectors must be the same size or the system will return an error.

Identity Matrix

An "identity matrix" is defined as a matrix which, when multiplied to another matrix A, produces the same matrix A. It is analogous to a 1 in normal arithmetic. For example, if I stands for an identity matrix, then A = I * A and also A = A * I. In order for an identity matrix to exist at all, A must be a square matrix (e.g., it must have the same number of columns as rows).

As it turns out, all identity matrices have the same form. They are square and consist of 1's along the main diagonal, and 0's everywhere else. For example, if A is a 3x3 matrix, then the identity matrix for A is:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

For a 4x4 matrix, I would be:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Since identity matrices are used frequently in matrix arithmetic, BASIC has a special function (IDN) that turns a square matrix into an identity matrix. For instance:

```
10  OPTION BASE 1
20  DIM I(2,2)
30  MAT I=IDN
      .
      .
```

The matrix I now contains the elements:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

If I was not a square matrix, line 20 would have returned an error.

Inverse Matrix

Although division is not defined for matrices, there is a similar operation which involves finding the inverse of a matrix. As with identity matrices, a matrix must be square in order to have an inverse. Inverse matrices are notated by a superscript -1. If A is a square matrix, then A^{-1} denotes its inverse. The inverse is defined by the equation:

$$A * A^{-1} = I$$

where I is the identity matrix. You can see how similar this is to division since, if A were a real number, then:

$$A * (1/A) = 1$$

The inverse of a matrix is found by using the INV function. For instance, the inverse of:

$$\mathbf{A}\begin{bmatrix} 0 & 2 & 0 \\ -1 & 2 & 0 \\ 2 & 0 & 2 \end{bmatrix}$$

is found by executing:

```
MAT A_inv=INV(A)
```

The system computes the values of the inverse and places them in the matrix `A_inv`:

$$\mathbf{A\_inv}\begin{bmatrix} 1 & -1 & 0 \\ .5 & 0 & 0 \\ -1 & 1 & .5 \end{bmatrix}$$

To check that this is really the inverse, you could execute the statement:

```
MAT B= A*A_inv
```

As expected, `B` turns out to be an identity matrix:

$$\mathbf{B}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Unfortunately, these expectations are not always fulfilled. Some matrices do not have an inverse. In other words, for a certain matrix called `A`, there exists no other matrix that, when multiplied to `A` produces an identity matrix. Matrices that don't have an inverse are called "singular". Singular matrices are easily detected and therefore aren't too dangerous. A more troublesome type of matrix is one that is "ill-conditioned". Ill-conditioned matrices are ones whose inverse can't be found by the computer because of round-off errors. These are difficult to detect and almost impossible to correct. We'll talk more about singular and ill-conditioned matrices, but before we do, we should discuss why you'd use an inverse in the first place.

Solving Simultaneous Equations

One of the most common applications of matrices is in the solution of simultaneous equations. Suppose we have the three equations shown below:

$$4X+2Y-Z=5$$
$$2X-3Y+3Z=5$$
$$X+Y-2Z=-3$$

Note that there are three unknowns (X,Y, and Z) and three equations. This is a necessity for solving by matrix arithmetic: you must have the same number of equations as unknowns. We can re-write these equations in matrix format as the product of two arrays:

$$\begin{bmatrix} 4 & 2 & -1 \\ 2 & -3 & 3 \\ 1 & 1 & -2 \end{bmatrix} * \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 5 \\ 5 \\ -3 \end{bmatrix}$$

For the sake of simplicity, let's name these three arrays A, B, and C. The equation, therefore, is:

$$A*B=C$$

If we multiply both sides of the equation by the inverse of A, we get:

$$A^{-1}*A*B=A^{-1}*C$$

Since $A^{-1}*A$ is simply a 3x3 identity matrix, the equation simplifies to:

$$I*B=A^{-1}*C$$

which further simplifies to:

$$B=A^{-1}*C$$

Remember, B is the matrix that contains the three variables X,Y and Z. To solve for these variables, therefore, all we have to do is multiply the matrix C by A^{-1}. This is accomplished in the program lines listed below.

```
        .
        .
200   DIM Solution(3),A_inv(3,3)
220   MAT A_inv=INV(A)
230   MAT Solution=A_inv*C
240   PRINT "X=";Solution(1)
250   PRINT "Y=";Solution(2)
260   PRINT "Z=";Solution(3)
        .
        .
```

When we run this program, it will print the values of X, Y, and Z. The values are:

```
X=1
Y=2
Z=3
```

For any set of simultaneous equations where there are the same number of unknown variables as there are equations, there are three possible classes of solution.

- There is no solution (e.g., there exist no values for the variables such that all of the equations are true).
- There are an infinite number of solutions.
- There is one, and only one, solution.

The first two cases are called "singular" sets of equations. You may recall that a singular matrix is one that has no inverse. It should not be surprising, therefore, that singular sets of equations always result in singular matrices when they are translated to matrix form. This is explained in the next section.

Singular Matrices

Any set of equations that has no solution or an infinite number of solutions is singular. Likewise, the matrix formed from these equations is also singular. More specifically, we mean the matrix on the left-hand side of the equation, what we've been calling matrix A. Consider the two equations listed below:

$$4X+6Y=5$$
$$4X+6Y=6$$

Obviously, there is no solution to this set of equations because any values assigned to X and Y will make only one of the equations true, not both. It is important to realize, however, that the singularity of these equations has nothing to do with the values on the right hand side of the equation. If, for example, we made the two equations the same,

$$4X+6Y=6$$
$$4X+6Y=6$$

then there would be an infinite number of solutions. For instance, X could equal 0 and Y equal 1, or X could equal 1.5 and Y could equal 0. In fact, so long as $X=1.5(1-Y)$, the two equations will always be true. What is important here is that the two equations,

$$4X+6Y=$$
$$4X+6Y=$$

will be singular regardless of what we put on the right-hand side of the equal sign. If we translate these equations into matrix form, we get:

$$\begin{bmatrix}4 & 6\\ 4 & 6\end{bmatrix} * \begin{bmatrix}X\\ Y\end{bmatrix}$$

The matrix,

$$\begin{bmatrix}4 & 6\\ 4 & 6\end{bmatrix}$$

is singular: it has no inverse. If, however, we call this matrix A and do an INV on it, the system will not report an error. On the contrary, it will go ahead and find what it thinks is an inverse. However, whatever matrix it comes up with will not be the inverse. Let's see what happens with our singular matrix A.

```
MAT A_inv=INV(A)
PRINT A_inv(*)
```

When we execute these statements, the system will display the following:

```
 .666666666667   .166666666667    0    -1
```

Arranging these values in the proper rows and columns, we get:

A_inv

$$\begin{bmatrix}.66666666667 & 0\\ .16666666667 & -1\end{bmatrix}$$

To see whether this is a real inverse, we can multiply it by A. If it is the inverse, the product should be an identity matrix.

```
MAT I=A*A_inv
```

I

$$\begin{bmatrix} 3.33333333333 & 5 \\ -4 & -6 \end{bmatrix}$$

Obviously, the system has made a mistake — A_inv is not the inverse of A. So how do we know if an inverse is valid? Or, to put it another way, how do we detect a singular matrix? We have just seen one method: multiply the matrix by its inverse and see whether you get an identity matrix. There is, however, a much easier method. You simply look at the "determinant" of the matrix.

The Determinant of a Matrix

The determinant of a matrix is defined somewhat mysteriously as the sum of all possible products formed by taking one element from each row in order starting from the top and one element from each column, where the sign of each product depends on the permutation of the column indices.

It's not really important that you understand how to calculate a determinant since the computer does it for you whenever you use the DET function. For instance, to print the determinant of matrix A, you would write:

```
PRINT DET(A)
```

Also the determinant is a byproduct of inversions. Thus, whenever you invert a matrix, the system computes the determinant and stores it. If you use DET without specifying a matrix, the system will return the determinant of the matrix most recently inverted. For example,

```
MAT A_inv=INV(A)
PRINT DET
```

would print the determinant of A.

Although the computation of the determinant is quite complex, its significance is very simple. If the determinant of a matrix equals 0, either a REAL underflow occurred during the inversion or **the matrix is singular**. To find out if an inversion is invalid, therefore, you merely test the matrix's determinant. If the determinant is zero, then the inverse is invalid. For example, if A is a square matrix, we could execute:

```
        .
        .
100   MAT A_inv=INV(A)
110   IF DET=0 THEN Singular
        .
        .
```

If A is singular, program control is passed to a line named "Singular". Note that we did not have to specify a matrix in line 110 since A was the last matrix inverted.

Unless you know for certain that a matrix is not singular, we recommend that you use the determinant test after each inversion. Otherwise, you may perform calculations using an invalid inverse.

Ill-Conditioned Matrices

In a few unusual cases, the inverse of a matrix will be invalid even though the determinant of the matrix is non-zero. These situations occur due to round-off errors internal to the computer. They are difficult to detect and even more difficult to correct. Fortunately, they occur very rarely. Unless you are having problems with a program involving matrix operations producing unexpected results, you can skip over to "Miscellaneous Matrix Functions." If you **are** having problems, listed below is an example of an ill-conditioned set of equations.

$$X_1 + 0X_2 + 3X_3 + 8X_4 = 12$$
$$2X_1 + X_2 + 6X_3 + 15.9X_4 = 24.9$$
$$3X_1 + X_2 + 8.9X_3 + 24X_4 = 36.9$$
$$4X_1 + X_2 + 11.9X_3 + 32X_4 = 48.9$$

We have selected the numbers on the right-hand side of the equation so that all of the X's equal 1. Watch what happens though when we try to solve these equations through matrix inversion. First, we set up the equations in matrix format.

$$\underset{\text{A}}{\begin{bmatrix} 1 & 0 & 3 & 8 \\ 2 & 1 & 6 & 15.9 \\ 3 & 1 & 8.9 & 24 \\ 4 & 1 & 11.9 & 32 \end{bmatrix}} * \underset{\text{Var}}{\begin{bmatrix} X1 \\ X2 \\ X3 \\ X4 \end{bmatrix}} = \underset{\text{Ans}}{\begin{bmatrix} 12 \\ 24.9 \\ 36.9 \\ 48.9 \end{bmatrix}}$$

Then we execute the program statements below.

```
          .
          .
100     MAT A_inv=INV(A)
110     MAT Var=A_inv*Ans
120     FOR Y=1 TO 4
130       PRINT Using """X(""",D,""")=""",K";Y,Var(Y)
140     NEXT Y
          .
          .
```

The computer displays:

```
X(1)=256
X(2)=0
X(3)=-32
X(4)=-16
```

Obviously something has gone wrong. The problem is that the inverse found by the computer is far off the mark from the actual inverse. The system of equations, though, is not singular. The determinant, though small, does not equal zero.

Detecting Ill-conditioned Matrices

Now that you've seen how ill-conditioning can affect the solutions to a set of simultaneous equations, you're probably wondering how you can tell an ill-conditioned matrix when you see one. There are a number of different techniques, none of which is entirely fail-proof. Used together, however, they are quite dependable.

In general, the determinant of an ill-conditioned matrix is very small compared with the elements of the matrix. So one of the first steps you can take is to look at the determinant. The term "very small" is, of course, relative. If a matrix contained elements all greater than 1000, then a determinant that equaled 10 would be very small. On the other hand, if all the elements in an array were less than 20 then a determinant of 10 would be quite reasonable. One equation for determining whether the determinant is "too small" is given below:

$$\frac{DET(A)}{\sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} A_{ij}^2}} << 1$$

We can execute this equation in a program as follows.

```
        .
        .
100 FOR X=(BASE A,1) TO (SIZE A,1)+(BASE A,1)-1
120   FOR Y=(BASE A,2) TO (SIZE A,2)+(BASE A,2)-1
130     Total=Total+A(X,Y)^2
140   NEXT Y
150 NEXT X
160 Test=DET(A)/SQR(Total)
170 IF Test<.001 THEN Ill_con
        .
        .
```

Note that line 170 can be changed depending on how much accuracy you require for your particular application. If we execute this program for the ill-conditioned matrix discussed earlier, the value of "Test" comes out to 9.527E-19. Since this value is much smaller than .001, this test would have correctly identified A as an ill-conditioned matrix.

Another technique for detecting ill-conditioned matrices is to multiply the matrix by its inverse and compare the product with the identity matrix. Again, you can demand as much accuracy as necessary. In the program below, we look for any elements in the product that differ by more than .001 from the identity matrix.

```
        .
        .
100   MAT I=IDN
110   MAT A_inv=INV(A)
120   MAT Product=A_inv*A
130   MAT Differ=(Product-I)
140   MAT Compare=Differ>(.001)
150   MAT Compare1=Differ<(-.001)
160   IF SUM(Compare)+SUM(Compare1)>0 THEN Ill_con
        .
        .
```

Applying this algorithm to our ill-conditioned matrix, we get:

A*A_inv

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & -.5 & 0 \\ -2 & -.5 & -4 & -16 \\ -2 & -.5 & -8 & -16 \end{bmatrix}$$

As you can see, 12 of the 16 elements differ from the identity matrix by more than 1, so this test also would have worked.

One drawback of this method is that it requires several additional matrices. If you are strapped for memory, this method could be unsatisfactory.

A third technique is to take the inverse of the inverse and compare it to the original matrix. The program below utilizes this method. Again, we are looking for differences greater than 0.001.

```
        .
        .
100   MAT A_inv=INV(A)
120   MAT A_inv_inv=INV(Ainv)
130   MAT Differ=(A_inv_inv-A)
140   MAT Compare=Differ>(.001)
150   MAT Compare1=Differ<(-.001)
160   IF SUM(Compare)+SUM(Compare1)>0 THEN Ill_con
        .
        .
```

Applying this technique to our ill-conditioned matrix, we find that all 16 elements of `A_inv_inv` differ from `A` by more than .001.

This technique will in general find more ill-conditioned matrices than the previous one. This is because any round-off errors are exaggerated by the second inverse. By the same token, it will occasionally detect an ill-conditioned matrix which might actually have been alright before the second inverse.

As stated before, none of these methods alone is decisive. What it all comes down to is that the precision of MAT INV falls off as a matrix approaches singularity. By using combinations of the tests described above, it is possible to determine how much precision has been lost, and then compare it to the precision actually required by your application.

Miscellaneous Matrix Functions

There are a few matrix functions that we haven't discussed yet. One of these is the transpose function (TRN). The transpose of a matrix is derived by exchanging rows for columns and columns for rows. If A is the matrix below,

$$\mathbf{A}\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix}$$

then,

```
MAT B=TRN(A)
```

would result in:

$$\mathbf{B}\begin{bmatrix}1 & 4 & 7\\ 2 & 5 & 8\\ 3 & 6 & 9\end{bmatrix}$$

A matrix does not have to be square to have a transpose. If A is,

$$\mathbf{A}\begin{bmatrix}0 & 1 & 8 & 3\\ 2 & 9 & 7 & -2\end{bmatrix}$$

then,

```
MAT B=TRN(A)
```

would result in:

$$\mathbf{B}\begin{bmatrix}0 & 2\\ 1 & 9\\ 8 & 7\\ 3 & -2\end{bmatrix}$$

The result array cannot be the same as the array being transposed. For example,

```
MAT A=TRN(A)
```

is an illegal statement and will cause an error.

The transpose function is useful for manipulating tables of data. It also has special significance for a small set of matrices called "orthogonal" matrices. An orthogonal matrix is defined as one whose transpose and inverse are the same.

Summing Rows and Columns

BASIC has two functions that return the sum of all rows and columns in an array. The totals are stored in a vector which you must dimension beforehand. Let A be the matrix shown below.

$$A = \begin{bmatrix} 3 & 6 & 18 & 7 \\ 1 & 0 & 41 & 2 \\ 4 & 3 & 12 & 11 \end{bmatrix}$$

If we execute:

```
10   OPTION BASE 1
20   DIM Row_sum(3),Col_sum(4)
30   MAT Row_sum=RSUM(A)
40   MAT Col_sum=CSUM(A)
50   PRINT "The sum of rows is";Row_sum(*);
60   PRINT "The sum of columns is";Col_sum(*);
70   END
```

The system will display:

```
The sum of rows is  34  44  30
The sum of columns is  8  9  71  20
```

Using Arrays for Code Conversion

Suppose you have an input device that provides information in 8-bit ASCII code. On the other hand, an output device in the same system uses a non-ASCII specialized 8-bit code. Examples might include specialized instrumentation, typesetting equipment, or a multitude of other devices. For each ASCII character, there is a corresponding code for the output device. There may be some ASCII characters (such as control characters) that are not to be converted. Let us assume that a null character (all bits set to zero) is used for those special characters. Here is how a conversion array is set up:

1. First, an array is created with 256 elements (0 thru 255). Each element address corresponds to the 8-bit INTEGER numeric equivalent of the ASCII character code. The contents of a given array element contains the output code for the corresponding ASCII input code. The array can be REAL or INTEGER. Usually, it is more efficient to use INTEGER arrays for converting 16-bit or shorter codes. The array must be filled by individual program statements (assignments or DATA and READ statements), or it can be filled from a mass storage file. If a file is used, the data must be created by some prior means. Fixed conversion codes can sometimes be generated by an algorithm in the introductory part of the program that performs the conversions.
2. Input data is placed in a string variable (see Chapter 5 for string variables techniques). Characters are then picked off, one character at a time, for conversion. Refer to *BASIC Interfacing Techniques* for more information about output operations.

Here is an example of how such an operation could be implemented:

```
1000    INTEGER Convert(0:255)
1010    DIM In$[80]
1020    Source=18       ! Source device selector
1030    Dest=22         ! Destination device selector
          .
          .
          .
    Initialize the conversion array here.
          .
          .
          .
2470    ENTER Source;Input$    ! Input line of ASCII
2480    FOR I=1 TO LEN(In$)    ! Send converted bytes
2490      OUTPUT Dest;CHR$(Convert(NUM(In$[I,I])));
2500    NEXT I
```

Note that the semicolon in line 2490 prevents sending a carriage-return and line-feed character pair at the end of each output line. This is usually necessary to prevent unwanted behavior when using ASCII strings to output non-ASCII data. This technique can be applied to arbitrary data conversions with virtually no limitations.

It is also possible to handle code conversions automatically in OUTPUT statements with the CONVERT options of the ASSIGN statement. See the ASSIGN Attributes discussion in BASIC Interfacing Techniques.

For further information about the topics in this chapter, see:

1 Coonen, Jerome T.; "An Implementation Guide to the Proposed Floating Point Standard", Computer Magazine, Jan. 1980.
2 Cody, William J. Jr. and William Waite; Software Manual for the Elementary Functions, Prentice Hall, 1980.
3 Sterbenz, Pat H.; Floating Point Computation, Prentice Hall, 1974.
4 Signum Newsletter, Oct. 1979.

Notes

String Manipulation

Chapter 5

Introduction

It is often desirable to store non-numerical information in the computer. A word, a name or a message can be stored in the computer as a **string**. Any sequence of characters may be used in a string. Quotation marks are used to delimit the beginning and ending of the string. The following are valid string assignments.

```
LET  A$="COMPUTER"
Fail$="The test has failed."
File_name$="INVENTORY"
Test$=Fail$[5,8]
```

The left-hand side of the assignment (the variable name) is equated to the right-hand side of the assignment (the literal).

String variable names are identical to numeric variable names with the exception of a dollar sign ($) appended to the end of the name.

The **length** of a string is the number of characters in the string. In the previous example, the length of A$ is 8 since there are eight characters in the literal "COMPUTER".

BASIC allows the dimensioned length of a string to range from 1 to 32 767 characters and the current length (number of characters in the string) to range from zero to the dimensioned length. A string of zero characters is often called a null string or an empty string.

The default dimensioned length of a string is 18 characters. The DIM, COM, and ALLOCATE statements are used to define string lengths up to the maximum length of 32 767 characters. An error results whenever a string variable is assigned more characters than its dimensioned length.

A string may contain any character. The only special case is when a quotation mark needs to be in a string. Two quotes, in succession, will embed a quote within a string.

```
10  Quote$="The time is ""NOW""."
20  PRINT Quote$
30  END
```

Produces: `The time is "NOW".`

String Storage

Strings whose length exceeds the default length of 18 characters must have space reserved before assignment. The following statements may be used.

- `DIM Long$[400]` Reserve space for a 400 character string.
- `COM Line$[80]` Reserve an 80 character common variable.
- `ALLOCATE Search$[Length]` Dynamic length allocation.

The maximum length of any string must not exceed 32 767 characters. A string may also be dimensioned to a length less than the default length of 18 characters.

The DIM statement reserves storage for strings.

```
DIM Part_number$[10],Decription$[64],Cost$[5]
```

The COM statement defines common variables that can be used by subprograms.

```
COM Name$[40],Phone$[14]
```

The ALLOCATE statement allows dynamic allocation of string storage. When the maximum length of a string cannot be determined ahead of time, the ALLOCATE statement can be used to reserve enough memory space for the string without wasting space.

```
ALLOCATE Line$[Length]
```

Strings that have been dimensioned but not assigned return the null string.

String Arrays

Large amounts of text are easily handled in arrays. For example:

```
DIM File$(1000)[80]
```

This statement reserves storage for 1000 lines of 80 characters per line. Do not confuse the brackets, which define the length of the string, with the parentheses which define the number of strings in the array. Each string in the array can be accessed by an index. For example:

```
PRINT File$(27)
```

Prints the 27th element in the array. Since each character in a string uses one byte of memory and each string in the array requires as many bytes as the length of the string, string arrays can quickly use a lot of memory.

A program saved on a disc as an ASCII type file can be entered into a string array, manipulated, and written back out to disc.

Evaluating Expressions Containing Strings

Evaluation Hierarchy

Evaluation of string expressions is simpler than evaluation of numerical expressions. The three allowed operations are extracting a substring, concatenation, and parenthesization. The evaluation hierarchy is presented in the following table.

Order	Operation
High	Parentheses
-	Substrings and Functions
Low	Concatenation

String Concatenation

Two separate strings are joined together by using the concatenation operator "&". The following program combines two strings into one.

```
10 One$="WRIST"
20 Two$="WATCH"
30 Concat$=One$&Two$
40 PRINT One$,Two$,Concat$
50 END
```

Prints:

```
WRIST      WATCH      WRISTWATCH
```

The concatenation operation, in line 30, appends the second string to the end of the first string. The result is assigned to a third string. An error results if the concatenation operation produces a string that is longer than the dimensioned length of the string being assigned.

Relational Operations

Most of the relational operators used for numeric expression evaluation can also be used for the evaluation of strings.

The following examples show some of the possible tests.

```
"ABC" = "ABC"                    True
"ABC" = " ABC"                   False

"ABC" < "AbC"                    True
"6" > "7"                        False
"2" < "12"                       False

"long" <= "longer"               True
"RE-SAVE" >= "RESAVE"            False
```

Any of these relational operators may be used: <, >, <=, >=, =, <>.

Testing begins with the first character in the string and proceeds, character by character, until the relationship has been determined.

The outcome of a relational test is based on the characters in the strings not on the length of the strings. For example:

```
"BRONTOSAURUS" < "CAT"
```

This relationship is true since the letter "C" is higher in ASCII value than the letter "B".

Note

When LEX is loaded, the outcome of a string comparison is based on the character's lexical value rather than the character's ASCII value. See the LEXICAL ORDER IS statement later in this chapter for more details.

Substrings

A subscript can be appended to a string variable name to define a **substring**. A substring may comprise all or just part of the original string. Brackets enclose the subscript which can be a constant, variable, or numeric expression. For instance:

String$[4] Specifies a substring starting with the fourth character of the original string.

The subscript must be in the range: 1 to the dimensioned length of the string plus 1. Note that the brackets now indicate the substring's starting position instead of the total length of the string as when reserving storage for a string.

Subscripted strings may appear on either side of the assignment.

Single-Subscript Substrings

When a substring is specified with only one numerical expression, enclosed with brackets, the expression is evaluated and rounded to an integer indicating the position of the first character of the substring within the string.

The following examples use the variable A$ which has been assigned the literal "DICTIONARY".

Statement	Output
PRINT A$	DICTIONARY
PRINT A$[0]	(error)
PRINT A$[1]	DICTIONARY
PRINT A$[5]	IONARY
PRINT A$[10]	Y
PRINT A$[11]	(null string)
PRINT A$[12]	(error)

When a single subscript is used it specifies the starting character position, within the string, of the substring. An error results when the subscript evaluates to zero or greater than the current length of the string plus 1. A subscript that evaluates to 1 plus the length of the string returns the null string ("") but does not produce an error.

Double-Subscript Substrings

A substring may have two subscripts, within brackets, to specify a range of characters. When a comma is used to separate the items within brackets, the first subscript marks the beginning position of the substring, while the second subscript is the ending position of the substring. The form is: A$[Start,End]

"JABBERWOCKY"[4,6] Specifies the substring: "BER"

When a semicolon is used in place of a comma, the first subscript again marks the beginning position of the substring, while the second subscript is now the length of the substring. The form is: A$[Start;Length]

"JABBERWOCKY"[4;6] Specifies the substring: "BERWOC"

In the following examples the variable B$ has been assigned to the literal "ENLIGHTENMENT":

Statement	Output
PRINT B$	ENLIGHTENMENT
PRINT B$[1,13]	ENLIGHTENMENT
PRINT B$[1;13]	ENLIGHTENMENT
PRINT B$[1,9]	ENLIGHTEN
PRINT B$[1;9]	ENLIGHTEN
PRINT B$[3,7]	LIGHT
PRINT B$[3;7]	LIGHTEN
PRINT B$[13,13]	N
PRINT B$[13;1]	N
PRINT B$[13,26]	(error)
PRINT B$[13;13]	(error)
PRINT B$[14;1]	(null string)

An error results if the second subscript in a comma separated pair is greater than the current string length plus 1 **or** if the sum of the subscripts in a semicolon separated pair is greater than the current string length plus 1.

Specifying the position just past the end of a string returns the null string.

Special Considerations

All substring operations allow a subscript to specify the first position past the end of a string. This allows strings to be concatenated without the concatenation operator. For instance:

```
10    A$="CONCAT"
20    A$[7]="ENATION"
30    PRINT A$
40    END
```

Produces: `CONCATENATION`

The substring assignment is only valid if the substring already has characters up to the specified position. Access beyond the first position past the end of a string results in the error:

```
ERROR 18  String ovfl, or substring err
```

A good practice is to dimension all strings including those shorter than the default length of eighteen characters.

Some very interesting assignments can be attempted. For example, a 14-character string can be assigned to a 3-character substring.

```
10    Big$="Too big to fit"
20    Small$="Little string"
30    !
40    Small$[1,3]=Big$
50    !
60    PRINT Small$
70    END
```

Prints: `Tootle string`

When a substring assignment specifies fewer characters than are available, any extra trailing characters are truncated.

The alternate assignment is shown in the next example. Here a 4-character string is assigned to a 8-character substring.

```
10    Big$="A large string"
20    Small$="tiny"
30    !
40    Big$[3,10]=Small$
50    !
60    DISP Big$
70    END
```

Prints: `A tiny     ring`

Since the subscripted length of the substring is greater than the length of the replacement string, enough blanks (ASCII spaces) are added to the end of the replacement string to fill the entire specified substring.

String-Related Functions

Several intrinsic functions are available in BASIC for the manipulation of strings. These functions include conversions between string and numeric values.

String Length

The "length" of a string is the number of characters in the string. The LEN function returns an integer whose value is equal to the string length. The range is from 0 (null string) thru 32 767. For example:

```
PRINT LEN("HELP ME")
```

Prints: 7

The following example program prints the length of a string that is typed on the keyboard.

```
10    DIM In$[160]
20    INPUT In$
30    Length=LEN(In$)
40    DISP Length;"characters in """;In$;""""
50    END
```

Try finding the length of a string containing only spaces. When the INPUT statement is used, any leading or trailing spaces are removed from items typed on the keyboard. Change INPUT to LINPUT in line 20 to allow leading and trailing spaces to be entered.

Substring Position

The "position" of a substring within a string is determined by the POS function. The function returns the value of the starting position of the substring or zero if the entire substring was not found. For instance:

```
PRINT POS("DISAPPEARANCE","APPEAR")
```

Prints: 4

The following example prints the positions of substrings found within a string.

```
10    DIM Sentence$[40],Word$(1:6)[8]
20    DATA CAT,ON,A,HOT,TIN,NATION
30    READ Word$(*)
40    Sentence$="WHERE IS THE CAT IN CONCATINATION"
50    !
60    FOR I=1 TO 6
70      Position=POS(Sentence$,Word$(I))   ! <- POS function
80      IF Position THEN
90        PRINT Sentence$
100       PRINT TAB(Position);Word$(I);TAB(35);"is at ";Position
110       PRINT
120     ELSE
130       PRINT "'";Word$(I);"' was not found"
140       PRINT
150     END IF
160   NEXT I
170   END
```

If POS returns a non-zero value, the entire substring occurs in the first string and the value specifies the starting position of the substring.

Note that POS returns the first occurrence of a substring within a string. By adding a subscript, and indexing through the string, the POS function can be used to find all occurrences of a substring. The following program uses this technique to extract each word from a sentence.

```
10    DIM A$[80]
20    A$="I know you think you understand what I said, but you don't."
30    INTEGER Scan,Found
40    Scan=1                              ! Current substring position
50    PRINT A$
60    REPEAT
70      Found=POS(A$[Scan]," ")           ! Find the next ASCII space
80      IF Found THEN
90        PRINT A$[Scan,Scan+Found-1]     ! Print the word
100       Scan=Scan+Found                 ! Adjust "Scan" past last match
110     ELSE
120       PRINT A$[Scan]                  ! Print last word in string
130     END IF
140   UNTIL NOT Found
150   END
```

As each occurrence is found, the new subscript specifies the remaining portion of the string to be searched.

String-to-Numeric Conversion

The VAL function converts a string expression into a numeric value. The string must evaluate to a valid number or error 32 will result.

```
ERROR 32 String is not a valid number
```

The number returned by the VAL function will be converted to and from scientific notation when necessary. For example:

```
PRINT VAL("123.4E3")
```

Prints: `123400`

The following program converts a fraction into its equivalent decimal value.

```
10    INPUT "Enter a fraction  (i.e. 3/4)",Fraction$
20    !
30    ON ERROR GOTO Err
40      Numerator=VAL(Fraction$)
50      !
60      IF POS(Fraction$,"/") THEN
70        Delimiter=POS(Fraction$,"/")
80        Denominator=VAL(Fraction$[Delimiter+1])
90      ELSE
100       PRINT "Invalid fraction"
110       GOTO Quit
120     END IF
130     !
140     PRINT Fraction$;" = ";Numerator/Denominator
150     GOTO Quit
160 Err: PRINT "ERROR Invalid fraction"
170      OFF ERROR
180 Quit:   END
```

Similar techniques can be used for converting: feet and inches to decimal feet or hours and minutes to decimal hours.

The NUM function converts a single character into its equivalent numeric value. The number returned is in the range: 0 to 255. For example:

```
PRINT NUM("A")
```

Prints: 65

The next program prints the value of each character in a name.

```
10    INPUT "Enter your first name",Name$
20    !
30    PRINT Name$
40    PRINT
50    FOR I=1 TO LEN(Name$)
60      PRINT NUM(Name$[I]);  ! Print value of each character
70    NEXT I
80    PRINT
90    END
```

Entering the name: JOHN will produce the following.

```
74   79   72   78
```

Numeric-to-String Conversion

The VAL$ function converts the value of a numeric expression into a character string. The string contains the same characters (digits) that appear when the numeric variable is printed. For example:

```
PRINT 1000000,VAL$(1000000)
```

Prints: 1.E+6 1.E+6

The next program converts a number into a string so the POS function can be used to seperate the mantissa from the exponent.

```
10    CONTROL 2,0;1 ! CAPS LOCK ON
20    INPUT "Enter a number with an exponent",Number
30    !
40    Number$=VAL$(Number)
50    !
60    PRINT Number$
70    E=POS(Number$,"E")
80    IF E THEN
90      PRINT "Mantissa is",Number$[1;E-1]
100     PRINT "Exponent is",Number$[E+1]
110   ELSE
120     PRINT "No exponent"
130   END IF
140   END
```

The CHR$ function converts a number into an ASCII character. The number can be of type INTEGER or REAL since the value is rounded, and a modulo 255 is performed. For example:

```
PRINT CHR$(97);CHR$(98);CHR$(99)
```

Prints: abc

The next program prints the values in the data statement as characters.

```
10    PRINT CHR$(12)  ! CLEAR SCREEN
20    PRINT CHR$(7)   ! RING THE BELL
30    !
40    DATA 34,130,89,111,117,32,103,111,116,32,105,116,33,128,34
50    INTEGER N(1:15)
60    READ N(*)
70    FOR I=1 TO 15
80      PRINT CHR$(N(I));
90    NEXT I
100   PRINT CHR$(7)
110   END
```

CRT Character Set

The following program prints the character set on the screen of the CRT.

```
10    ! Program: CRT Character Set.
20    !
30    PRINT CHR$(12);"CRT Character Set"
40    STATUS 1,9;Line_length ! 50 or 80 Column
50    Left=Line_length/2-16
60    !
70    FOR I=0 TO 255
80      Col=I MOD 16*2+Left
90      Row=I DIV 16+3
100     IF Col=Left THEN
110       PRINT TABXY(Left-5,Row);
120       PRINT USING "3D";I
130     END IF
140     PRINT TABXY(Col,Row);
150     CONTROL 1,4;1         ! Display Functions on
160     PRINT CHR$(I);        ! PRINT the Character
170     CONTROL 1,4;0         ! Display Functions off
180   NEXT I
190   PRINT
200   I=127
210   ON KNOB .08 GOSUB Change
220   DISP USING "5A,5D,X,2A,B,B";"ASCII",I,"=",128,I
230   GOTO 220
240 Change:   I=I-KNOBX/10
250           IF I<0 THEN I=0
260           IF I>255 THEN I=255
270           RETURN
280   END
```

ASCII character values from 128 to 159 are treated differently by different systems. Refer to the section "The Extended Character Set" found later in this chapter.

String Functions

String Reverse

The REV$ function returns a string created by reversing the sequence of characters in the given string.

```
PRINT REV$("Snack cans")
```

Prints: `snac kcanS`

A common use for the REV$ function is to find the last occurrence of an item in a string.

```
10    DIM List$[30]
20    List$="3.22 4.33 1.10 8.55 12.20 1.77"
30    Length=LEN(List$)
40    Last_space=POS(REV$(List$)," ")   ! "SPACE" is delimiter
50    DISP "The last item is:";List$[1+Length-Last_space]
60    END
```

Displays: `The last item is: 1.77`

String Repeat

The RPT$ function returns a string created by repeating the specified string, a given number of times.

```
PRINT RPT$("* *",10)
```

Prints: `* ** ** ** ** ** ** ** ** ** *`

Here is a short program that uses RPT$ to create an image for a formatted print statement.

```
10    Items=7
20    DATA 50,900,2,444,37,2001,32768
30    ALLOCATE Array(1:Items)
40    READ Array(*)
50    FOR I=1 TO Items
60      Digits=INT(1+LGT(Array(I)))
70      IF Digits>Maxdigits THEN Maxdigits=Digits
80    NEXT I
90    Form$="XX,"&RPT$("D",Maxdigits)&".DD"
100   PRINT "Using the image: ";Form$
110   PRINT USING Form$;Array(*)
120   END
```

Trimming a String

The TRIM$ function returns a string with all leading and trailing blanks (ASCII spaces) removed.

```
PRINT "*";TRIM$("     1.23     ");"*"
```

Prints: `*1.23*`

TRIM$ is often used to extract fields from data statements or keyboard input.

```
10    INPUT "Enter your full name",Name$
20    First$=TRIM$(Name$[1,POS(Name$," ")])
30    Last$=TRIM$(Name$[1+LEN(Name$)-POS(REV$(Name$)," ")])
40    PRINT Name$,LEN(Name$)
50    PRINT Last$,LEN(Last$)
60    PRINT First$,LEN(First$)
70    END
```

Note that the INPUT statement trims leading and trailing blanks from whatever is typed. If you need to enter leading or trailing spaces, use the LINPUT statement.

Case Conversion

The case conversion functions, UPC$ and LWC$, return strings with all characters converted to the proper case. UPC$ converts all lowercase characters to their corresponding uppercase characters and LWC$ converts any uppercase characters to their corresponding lowercase characters. Roman Extension characters will be converted according to the current lexical order. See the LEXICAL ORDER IS statement later in this chapter for the case conversion listings.

```
10    DIM Word$[160]
20    LINPUT "Enter a few characters",Word$
30    PRINT
40    PRINT "You typed: ";Word$
50    PRINT "Uppercase: ";UPC$(Word$)
60    PRINT "Lowercase: ";LWC$(Word$)
70    END
```

A more general character replacement method is obtained by using a buffer that was assigned an indexed conversion. Indexed conversion uses the incoming character's ASCII value as an index into a string of characters and returns the character in that position. In the following program, the conversion string is created in lines 30 and 50. The conversion string specifies all lowercase characters are to be replaced by their corresponding uppercase character.

```
10    DIM Cipher$[256],A$[80]
20    FOR I=1 TO 255                        ! Create conversion string
30      Cipher$=Cipher$&UPC$(CHR$(I))
40    NEXT I
50    Cipher$=Cipher$&UPC$(CHR$(0))
60    ASSIGN @F TO BUFFER [160];CONVERT OUT BY INDEX Cipher$
70    LOOP
80      INPUT A$
90      OUTPUT @F;A$                        ! Conversion occurs
100     ENTER @F;A$
110     PRINT A$
120   END LOOP
130   END
```

Searching and Sorting

Information stored in a string array often requires sorting. There are over a dozen common algorithms that may be used. Each alogrithm has certain advantages depending on the number of items to be sorted, the current order of the items, the time allowed to sort the items, and the complexity of the algorithm. One of the simplest (and most inefficient) sorts to implement is the "bubble" sort. The following program is a slight variation of the bubble sort.

```
10    ! Program: SORT
20    !
30    READ N
40    DATA 10   ! NUMBER OF ITEMS TO SORT
50    ALLOCATE Word$(N)[5],Temp$[5]
60    READ Word$(*)   ! READ ENTIRE ARRAY
70    DATA zero,one,two,three,four,five,six,seven,eight,nine,ten
80    PRINT Word$(*)
90    PRINT
100 Sort:FOR I=0 TO N-1
110         IF Word$(I)>Word$(I+1) THEN
120           Temp$=Word$(I)
130           Word$(I)=Word$(I+1)
140           Word$(I+1)=Temp$
150           GOTO Sort
160         END IF
170       NEXT I
180   PRINT Word$(*)
190   END
```

This example prints the contents of the array before and after sorting.

Before sorting:

```
zero        one         two         three       four        five
six         seven       eight       nine        ten
```

After sorting:

```
eight       five        four        nine        one         seven
six         ten         three       two         zero
```

The strings are sorted in ascending order. If the relational operator in line 110 is changed from the greater than sign ">" to the less than sign "<", the strings will be sorted in descending order.

MAT Functions and String Arrays

MAT functions (available with MAT) are commonly used to manipulate data in numeric arrays. However, several of these functions can be used with string arrays. For example, a string array is copied into another string array by the following.

```
MAT Copy$ = Original$
```

Note that only the variable name is necessary. The array specifier "(*)" need not be included when using the MAT statement.

Every element in a string array will be initialized to a constant value by the following statement.

```
MAT Array$ = (Null$)
```

The constant value can be a literal or a string expression and is enclosed in parentheses to distinguish it from being an array name.

A list of items can be sorted very quickly by the MAT SORT statement.

```
10    ! Program: SORT_LIST
20    DIM List$(1:5)[6]
30    DATA Bread,Milk,Eggs,Bacon,Coffee
40    READ List$(*)
50    !
60    PRINT "original order"
70    PRINT List$(*)
80    !
90    PRINT "ascending order"
100   MAT SORT List$(*)
110   PRINT List$(*)
120   !
130   PRINT "descending order"
140   MAT SORT List$(*) DES
150   PRINT List$(*)
160   END
```

Running this program produces:

```
original order
Bread     Milk      Eggs      Bacon     Coffee

ascending order
Bacon     Bread     Coffee    Eggs      Milk

descending order
Milk      Eggs      Coffee    Bread     Bacon
```

Sorting by Substrings

A substring range can be appended to the end of a MAT SORT statement. Items will then be sorted by the characters within the substring specified. No error results from specifying a substring position beyond the current length of the string.

```
10    PRINT CHR$(12)  ! Program: SUBSORT
20    DATA 1 OLD ORANGE,2 TINY TOADS,3 TALL TREES,4 FAT FOWLS,5 FRIED FISH
30    DATA 6 SLOW SNAILS,7 SLIMY SLUGS,8 AWFUL HOURS,9 NASTY KNIVES
40    DIM Thing$(1:9)[38]
50    READ Thing$(*)
60    First=1
70    Length=1
80    DISP "Use KNOB and SHIFT-KNOB to change sort field."
90    ON KNOB .15 GOTO Slide
100 Go:MAT SORT Thing$(*)[First;Length]
110   FOR I=1 TO 9
120     PRINT TABXY(10,I);Thing$(I);"    "
130   NEXT I
140 W:GOTO W
150   !
160 Slide:      !
170   S=SGN(KNOBY)
180   H=SGN(KNOBX)
190   IF S THEN
200     Length=Length+S*(S>0 AND Length<16)+S*(S<0 AND Length>1)
210     END IF
220   IF H THEN
230     First=First+H*(H>0 AND First<18)+H*(H<0 AND First>1)
240   END IF
250   DISP "MAT SORT Thing$(*)[";First;";";Length;"]"
260   PRINT TABXY(9,10);RPT$(" ",First);RPT$("^",Length);RPT$(" ",10)
270   GOTO Go
280   END
```

Adding Items to a Sorted List

Lists of strings can be maintained in sorted order. Every time a new item is added to the list, the list is sorted by the MAT SORT statement. To prevent overwriting any of the items already in the list, items should be added to the top (first array element) of a list sorted in ascending order and to the bottom (last array element) of a list sorted in descending order.

```
10    PRINT CHR$(12)
20    ! Since arrays are in COM, they "remember" old values.
30    ! After running, execute SCRATCH C to clear the arrays.
40    !
50    COM Ascend$(1:18)[18],Descend$(1:18)[18]
60 Again:I=I+1
70    INPUT "Enter a word",Word$
80    Ascend$(1)=Word$              ! Fill array at top
90    Descend$(18)=Word$            ! Fill array at bottom
100   CALL See
110   IF I<18 THEN Again
120   BEEP
130   END
140   !-------------------------------------------------
150   SUB See                       ! DISPLAY THE ARRAYS
160     COM Ascend$(*),Descend$(*)
170     MAT SORT Ascend$            ! <- ascending sort
180     MAT SORT Descend$ DES       ! <- descending sort
190     FOR J=1 TO 18
200       PRINT TABXY(1,J);RPT$(" ",49)
210       PRINT TABXY(1,J);J;TABXY(11,J);Ascend$(J);TABXY(31,J);Descend$(J)
220     NEXT J
230   SUBEND
```

Sorting by Multiple Keys

When sorting a multi-dimension array, it is possible to specify more than one key. The array will be sorted by the first key then the second key and so on until the key specifiers are exhausted. Once the first key sorts items into similar groups, the items within a group can be arranged in any order you choose.

```
10    COM Tool$(1:8,1:3)[10]
20    DATA PENCIL,RED,35,PENCIL,BLUE,12,PENCIL,GREEN,0,PENCIL,BLACK,17
30    DATA PEN,BLACK,17,PEN,BLUE,127,PEN,RED,55,PEN,GREEN,43
40    READ Tool$(*)
50    PRINT
60    PRINT "*** UNSORTED LIST ***"
70    Display
80    PRINT "*** SORT BY COLOR ***"
90    MAT SORT Tool$(*,2)[1,3]      ! Sort color by first three letters.
100   Display
110   PRINT "*** SORT BY COLOR THEN BY NAME ***"
120   MAT SORT Tool$(*,2),(*,1)     ! Two key sort.
130   Display
140   PRINT "*** SORT BY NAME THEN BY COLOR ***"
150   MAT SORT Tool$(*,1),(*,2)[1;3] DES
160   Display
170   END
180   !----------------------
190   SUB Display
200     COM Tool$(*)
210     K=K+1
220     FOR I=1 TO 8
230       FOR J=1 TO 3
240         PRINT Tool$(I,J),
250       NEXT J
260       PRINT
270     NEXT I
280   SUBEND
```

Sorting to a Vector

It is possible to determine the sorting order of items in an array without disturbing the array. This is accomplished by "sorting" to a single-dimensioned numeric array (vector). The vector will then contain the subscripts of the items in the order that the items would have been arranged.

```
10    DIM Month$(1:12)[3],Fix(1:12)
20    DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
30    READ Month$(*)
40    MAT SORT Month$ TO Fix     ! Sort to vector
50    PRINT Month$(*)
60    PRINT Fix(*)
70    FOR I=1 TO 12
80      PRINT Month$(Fix(I)),    ! Print months alphabetically
90    NEXT I
100   END
```

Running this program produces:

```
JAN   FEB   MAR   APR   MAY   JUN   JUL   AUG   SEP   OCT   NOV   DEC

4   8   12   2   1   7   6   3   5   11   10   9

APR   AUG   DEC   FEB   JAN   JUL   JUN   MAR   MAY   NOV   OCT   SEP
```

The first element of the vector contains a four (4), indicating the fourth element in the array would be the first element if the array were actually sorted.

Reordering an Array

The rows and columns of multiple dimension arrays can be reordered. Reordering is made according to a reorder vector (single dimension array). The vector contains the values of the subscripts of the array. When the array is reordered, the columns (or rows) are arranged according to the the order of the subscripts in the reorder vector. See the following program for an example of reordering.

```
10    PRINT CHR$(12);      ! SORT_DEMO
20    DIM Size$(0:1)[5],Color$(0:2)[5],Shape$(0:1)[5]
30    COM Ident$(0:3)[5],Array$(0:3,0:11)[6],Order(0:3),Field,Down
40    DATA COUNT,SIZE,COLOR,SHAPE
50    DATA small,large,blue,red,green,cube,ball,1,2,3,0
60    READ Ident$(*),Size$(*),Color$(*),Shape$(*),Order(*)
70    FOR I=0 TO 11
80      Array$(0,I)=RPT$(" ",I<9)&VAL$(I+1)
90      Array$(1,I)=Size$(I DIV 6)
100     Array$(2,I)=Color$(I DIV 2 MOD 3)
110     Array$(3,I)=Shape$(I MOD 2)
120   NEXT I
130   ON KBD CALL Do_key
140 Again:D$=" Ascending"
150   IF Down THEN D$="Descending"
160   DISP D$;" sort on field #";Field+1
170   Sort
180   Display
190   GOTO Again
200   END
210   !-----------------------------------------
220   SUB Display
230     COM Ident$(*),Array$(*),Order(*),Field,Down
240     PRINT TABXY(1,1);
250     PRINT "Press: A for ascending sort"
260     PRINT "       D for descending sort"
270     PRINT "       R to reorder array"
280     PRINT "     1-4 for sort field";TABXY(1,5)
290     PRINT USING "#,3X,5A";Ident$(*)
300     FOR I=0 TO 11
310       PRINT TABXY(1,I+7);
320       FOR J=0 TO 3
330         PRINT USING "#,3X,5A";Array$(J,I)
340       NEXT J
350     NEXT I
360   SUBEND
370   !-----------------------------------------
380   SUB Sort
390     COM Ident$(*),Array$(*),Order(*),Field,Down
400     IF Down THEN
410       MAT SORT Array$(Field,*) DES
420     ELSE
430       MAT SORT Array$(Field,*)
440     END IF
450   SUBEND
460   !-----------------------------------------
470   SUB Do_key
480     COM Ident$(*),Array$(*),Order(*),Field,Down
490     Key$=KBD$
500     SELECT Key$
510     CASE "1" TO "4"
520       Field=VAL(Key$)-1
530     CASE "A","a"
540       Down=0
550     CASE "D","d"
560       Down=1
```

Lexical Tables

The following tables show the five predefined lexical orders available with the LEXICAL ORDER IS statement.

Notation

All of the lexical tables use the following notation.

sequence number →	113
character displayed →	a
ASCII value →	(97)

Characters not available on the keyboard can be entered by pressing the ANY CHAR key and typing the value enclosed in parentheses (with leading zeros, if needed). The character will be collated according to the sequence number shown above the character.

ASCII Lexical Order

The ASCII lexical order uses the character's ASCII value as the sequence number. There are no special cases (mode table entries) used in the ASCII lexical order.

Case Conversions

The following lists show the UPC$ and LWC$ transformations for the ASCII lexical order.

UPC$

```
abcdefghijklmnopqrstuvwxyzçñâêôûáéóúàèòùãẽõũîìíøæìïβãđõšÿþ

ABCDEFGHIJKLMNOPQRSTUVWXYZÇÑÂÊÔÛÁÉÓÚÀÈÒÙÃẼÕŨÎÌÍØÆÌÏβÃĐÕŠŸÞ
```

LWC$

```
ABCDEFGHIJKLMNOPQRSTUVWXYZÀÂÈÊËÎÏÙÛÇÑÀØÆÃÕÜÉÔÁÃĐÍÌÓÒÕŠÚŸÞ

abcdefghijklmnopqrstuvwxyzàâèêëîïùûçñàøæãõüéôáãđíìóòõšúÿþ
```

Note

There are slight variations in the operation of the UPC$ and LWC$ functions depending on the lexical order in effect. In other words, the lexical order determines which character will be returned by the UPC$ and LWC$ functions. The case conversion lists show which characters should be expected for each lexical order. To simplify the lists, characters not affected have been excluded.

LEXICAL ORDER IS ASCII

Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.
0	NU	(0)	52	4	(52)	104	h	(104)	156	9C	(156)	208	À	(208)
1	SH	(1)	53	5	(53)	105	i	(105)	157	9D	(157)	209	Î	(209)
2	SX	(2)	54	6	(54)	106	j	(106)	158	9E	(158)	210	Ø	(210)
3	EX	(3)	55	7	(55)	107	k	(107)	159	9F	(159)	211	Æ	(211)
4	ET	(4)	56	8	(56)	108	l	(108)	160		(160)	212	à	(212)
5	EO	(5)	57	9	(57)	109	m	(109)	161	À	(161)	213	í	(213)
6	AK	(6)	58	:	(58)	110	n	(110)	162	Â	(162)	214	ø	(214)
7	BL	(7)	59	;	(59)	111	o	(111)	163	È	(163)	215	æ	(215)
8	BS	(8)	60	<	(60)	112	p	(112)	164	Ê	(164)	216	Ä	(216)
9	HT	(9)	61	=	(61)	113	q	(113)	165	Ë	(165)	217	ì	(217)
10	LF	(10)	62	>	(62)	114	r	(114)	166	Î	(166)	218	Ö	(218)
11	VT	(11)	63	?	(63)	115	s	(115)	167	Ï	(167)	219	Ü	(219)
12	FF	(12)	64	@	(64)	116	t	(116)	168	´	(168)	220	É	(220)
13	CR	(13)	65	A	(65)	117	u	(117)	169	`	(169)	221	ï	(221)
14	SO	(14)	66	B	(66)	118	v	(118)	170	^	(170)	222	ß	(222)
15	SI	(15)	67	C	(67)	119	w	(119)	171	¨	(171)	223	Ô	(223)
16	DL	(16)	68	D	(68)	120	x	(120)	172	~	(172)	224	Á	(224)
17	D1	(17)	69	E	(69)	121	y	(121)	173	Ù	(173)	225	Ã	(225)
18	D2	(18)	70	F	(70)	122	z	(122)	174	Û	(174)	226	ã	(226)
19	D3	(19)	71	G	(71)	123	{	(123)	175	£	(175)	227	Ð	(227)
20	D4	(20)	72	H	(72)	124	|	(124)	176	¯	(176)	228	đ	(228)
21	NK	(21)	73	I	(73)	125	}	(125)	177	B1	(177)	229	ƒ	(229)
22	SY	(22)	74	J	(74)	126	~	(126)	178	B2	(178)	230	Ì	(230)
23	EB	(23)	75	K	(75)	127	▒	(127)	179	·	(179)	231	Ó	(231)
24	CN	(24)	76	L	(76)	128	CL	(128)	180	Ç	(180)	232	Ò	(232)
25	EM	(25)	77	M	(77)	129	IV	(129)	181	ç	(181)	233	Õ	(233)
26	SB	(26)	78	N	(78)	130	BG	(130)	182	Ñ	(182)	234	õ	(234)
27	EC	(27)	79	O	(79)	131	IB	(131)	183	ñ	(183)	235	Š	(235)
28	FS	(28)	80	P	(80)	132	UL	(132)	184	¡	(184)	236	š	(236)
29	GS	(29)	81	Q	(81)	133	IU	(133)	185	¿	(185)	237	Ú	(237)
30	RS	(30)	82	R	(82)	134	BG	(134)	186	¤	(186)	238	Ÿ	(238)
31	US	(31)	83	S	(83)	135	IB	(135)	187	£	(187)	239	ÿ	(239)
32		(32)	84	T	(84)	136	WH	(136)	188	¥	(188)	240	Þ	(240)
33	!	(33)	85	U	(85)	137	RD	(137)	189	§	(189)	241	þ	(241)
34	"	(34)	86	V	(86)	138	YE	(138)	190	ƒ	(190)	242	F2	(242)
35	#	(35)	87	W	(87)	139	GR	(139)	191	¢	(191)	243	F3	(243)
36	$	(36)	88	X	(88)	140	CY	(140)	192	â	(192)	244	F4	(244)
37	%	(37)	89	Y	(89)	141	BU	(141)	193	ê	(193)	245	I0	(245)
38	&	(38)	90	Z	(90)	142	MG	(142)	194	ô	(194)	246	–	(246)
39	'	(39)	91	[	(91)	143	BK	(143)	195	û	(195)	247	¼	(247)
40	(	(40)	92	\	(92)	144	90	(144)	196	á	(196)	248	½	(248)
41	)	(41)	93	]	(93)	145	91	(145)	197	é	(197)	249	ª	(249)
42	*	(42)	94	^	(94)	146	92	(146)	198	ó	(198)	250	º	(250)
43	+	(43)	95	_	(95)	147	93	(147)	199	ú	(199)	251	«	(251)
44	,	(44)	96	`	(96)	148	94	(148)	200	à	(200)	252	■	(252)
45	–	(45)	97	a	(97)	149	95	(149)	201	è	(201)	253	»	(253)
46	.	(46)	98	b	(98)	150	96	(150)	202	ò	(202)	254	±	(254)
47	/	(47)	99	c	(99)	151	97	(151)	203	ù	(203)	255	▯	(255)
48	0	(48)	100	d	(100)	152	98	(152)	204	ä	(204)			
49	1	(49)	101	e	(101)	153	99	(153)	205	ë	(205)			
50	2	(50)	102	f	(102)	154	9A	(154)	206	ö	(206)			
51	3	(51)	103	g	(103)	155	9B	(155)	207	ü	(207)			

FRENCH Lexical Order

The FRENCH lexical order table contains two special entries. The hyphen character (-) is assigned as a "don't care" character and a "2 for 1" character replacement is made for the "ß" character.

```
ß = ss
```

A string containing the hyphen will match the same string without the hyphen and a string containing only a hyphen will match the null string. For example:

```
LEXICAL ORDER IS FRENCH
IF "RE-STORE"="RESTORE" THEN PRINT "TRUE"
```

Prints: TRUE

Case Conversions

The following lists show the UPC$ and LWC$ transformations for the FRENCH lexical order.

UPC$

```
abcdefghijklmnopqrstuvwxyzçñâêôûáéóúàèòùäëöüîàíøæìïßãđõšÿÞ

ABCDEFGHIJKLMNOPQRSTUVWXYZÇÑÂÊÔÛÁÉÓÚÀÈÒÙÄËÖÜÎÀÍØÆÌÏßÃĐÕŠŸÞ
```

LWC$

```
ABCDEFGHIJKLMNOPQRSTUVWXYZÀÂÈÊËÎÏÙÛÇÑÁØÆÃÖÜÉÔÁÃĐÍÌÓÒÕŠÚŸÞ

abcdefghijklmnopqrstuvwxyzàâèêëîïùûçñáøæãöüéôáãđíìóòõšúÿÞ
```

LEXICAL ORDER IS FRENCH

Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.
	-	(45)	51	4	(52)	81	P	(80)	113	l	(108)	153	½	(248)
0	NU	(0)	52	5	(53)	82	Q	(81)	114	m	(109)	154	ª	(249)
1	SH	(1)	53	6	(54)	83	R	(82)	115	n	(110)	155	º	(250)
2	SX	(2)	54	7	(55)	84	S	(83)	116	ñ	(183)	156	«	(251)
3	EX	(3)	55	8	(56)	85	Š	(235)	117	o	(111)	157	■	(252)
4	ET	(4)	56	9	(57)	86	T	(84)	117	ô	(194)	158	»	(253)
5	EO	(5)	57	:	(58)	87	U	(85)	117	ó	(198)	159	±	(254)
6	AK	(6)	58	;	(59)	87	Ù	(173)	117	ò	(202)	160	▒	(127)
7	BL	(7)	59	<	(60)	87	Û	(174)	117	ö	(206)	161		(160)
8	BS	(8)	60	=	(61)	87	Ü	(219)	117	ø	(214)	162	B1	(177)
9	HT	(9)	61	>	(62)	87	Ú	(237)	117	õ	(234)	163	B2	(178)
10	LF	(10)	62	?	(63)	88	V	(86)	118	p	(112)	164	F2	(242)
11	VT	(11)	63	@	(64)	89	W	(87)	119	q	(113)	165	F3	(243)
12	FF	(12)	64	A	(65)	90	X	(88)	120	r	(114)	166	F4	(244)
13	CR	(13)	64	À	(161)	91	Y	(89)	121	s	(115)	167	ID	(245)
14	SO	(14)	64	Â	(162)	91	Ÿ	(238)	121	ß	(222)	168	CL	(128)
15	SI	(15)	64	Ä	(208)	92	Z	(90)	122	š	(236)	169	IV	(129)
16	DL	(16)	64	Æ	(211)	93	Þ	(240)	123	t	(116)	170	BG	(130)
17	D1	(17)	64	Ä	(216)	94	[	(91)	124	u	(117)	171	IB	(131)
18	D2	(18)	64	Á	(224)	95	\	(92)	124	û	(195)	172	UL	(132)
19	D3	(19)	64	Ã	(225)	96	]	(93)	124	ú	(199)	173	IU	(133)
20	D4	(20)	65	B	(66)	97	^	(94)	124	ù	(203)	174	BG	(134)
21	NK	(21)	66	C	(67)	98	_	(95)	124	ü	(207)	175	IB	(135)
22	SY	(22)	66	Ç	(180)	99	`	(96)	125	v	(118)	176	WH	(136)
23	EB	(23)	67	D	(68)	100	a	(97)	126	w	(119)	177	RD	(137)
24	CN	(24)	68	Đ	(227)	100	â	(192)	127	x	(120)	178	YE	(138)
25	EM	(25)	69	E	(69)	100	á	(196)	128	y	(121)	179	GR	(139)
26	SB	(26)	69	È	(163)	100	à	(200)	128	ÿ	(239)	180	CY	(140)
27	EC	(27)	69	Ê	(164)	100	ä	(204)	129	z	(122)	181	BU	(141)
28	FS	(28)	69	Ë	(165)	100	å	(212)	130	þ	(241)	182	MG	(142)
29	GS	(29)	69	É	(220)	100	æ	(215)	131	{	(123)	183	BK	(143)
30	RS	(30)	70	F	(70)	100	ã	(226)	132	|	(124)	184	90	(144)
31	US	(31)	71	G	(71)	101	b	(98)	133	}	(125)	185	91	(145)
32		(32)	72	H	(72)	102	c	(99)	134	~	(126)	186	92	(146)
33	!	(33)	73	I	(73)	103	ç	(181)	135	´	(168)	187	93	(147)
34	"	(34)	73	Î	(166)	104	d	(100)	136	`	(169)	188	94	(148)
35	#	(35)	73	Ï	(167)	105	đ	(228)	137	^	(170)	189	95	(149)
36	$	(36)	73	Í	(229)	106	e	(101)	138	¨	(171)	190	96	(150)
37	%	(37)	73	Ì	(230)	106	ê	(193)	139	~	(172)	191	97	(151)
38	&	(38)	74	J	(74)	106	é	(197)	140	£	(175)	192	98	(152)
39	'	(39)	75	K	(75)	106	è	(201)	141	¯	(176)	193	99	(153)
40	(	(40)	76	L	(76)	106	ë	(205)	142	·	(179)	194	9A	(154)
41	)	(41)	77	M	(77)	107	f	(102)	143	¡	(184)	195	9B	(155)
42	*	(42)	78	N	(78)	108	g	(103)	144	¿	(185)	196	9C	(156)
43	+	(43)	79	Ñ	(182)	109	h	(104)	145	¤	(186)	197	9D	(157)
44	,	(44)	80	O	(79)	110	i	(105)	146	£	(187)	198	9E	(158)
45	.	(46)	80	Ø	(210)	110	î	(209)	147	¥	(188)	199	9F	(159)
46	/	(47)	80	Ö	(218)	110	í	(213)	148	§	(189)	200	K	(255)
47	0	(48)	80	Ô	(223)	110	ì	(217)	149	ƒ	(190)			
48	1	(49)	80	Ó	(231)	110	ï	(221)	150	¢	(191)			
49	2	(50)	80	Ò	(232)	111	j	(106)	151	-	(246)			
50	3	(51)	80	Õ	(233)	112	k	(107)	152	¼	(247)			

SPANISH Lexical Order

The SPANISH lexical order table contains five special entries. Four of these entries are "1 for 2" character replacements. When the following character pairs are found in a string, a single sequence number is used to represent the pair.

```
CH = 68     cH = 103
Ch = 68     ch = 103
LL = 78     lL = 113
Ll = 78     ll = 113
```

The remaining special case is a "2 for 1" entry for the "ß" character.

```
ß = ss
```

Case Conversions

The following lists show the UPC$ and LWC$ transformations for the SPANISH lexical order.

UPC$

```
abcdefghijklmnopqrstuvwxyzçñáéíóúàèòùäëöüîâíøæìïßãđõšÿþ

ABCDEFGHIJKLMNOPQRSTUVWXYZCÑAEOUAEOUAEOUÄËÖÜIÁIØÆIIßÃĐÕŠŸÞ
```

LWC$

```
ABCDEFGHIJKLMNOPQRSTUVWXYZÀÂÈÊËÎÏÙÛÇÑÄØÆÃÕÜÉÔÁÃĐÍÌÓÒŠÚÝÞ

abcdefghijklmnopqrstuvwxyzàâèêëîïùûçñäøæãõüéôáãđíìóoõšúýþ
```

LEXICAL ORDER IS SPANISH

Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.
0	NU	(0)	52	4	(52)	84	P	(80)	116	l	(108)	157	½	(248)
1	SH	(1)	53	5	(53)	85	Q	(81)	118	m	(109)	158	ª	(249)
2	SX	(2)	54	6	(54)	86	R	(82)	119	n	(110)	159	º	(250)
3	EX	(3)	55	7	(55)	87	S	(83)	120	ñ	(183)	160	«	(251)
4	ET	(4)	56	8	(56)	88	Š	(235)	121	o	(111)	161	■	(252)
5	EO	(5)	57	9	(57)	89	T	(84)	121	ô	(194)	162	»	(253)
6	AK	(6)	58	:	(58)	90	U	(85)	121	ó	(198)	163	±	(254)
7	Δ	(7)	59	;	(59)	90	Ù	(173)	121	ò	(202)	164	▒	(127)
8	BS	(8)	60	<	(60)	90	Û	(174)	121	ö	(206)	165		(160)
9	HT	(9)	61	=	(61)	90	Ü	(219)	121	ø	(214)	166	B1	(177)
10	LF	(10)	62	>	(62)	90	Ú	(237)	121	ð	(234)	167	B2	(178)
11	VT	(11)	63	?	(63)	91	V	(86)	122	p	(112)	168	F2	(242)
12	FF	(12)	64	@	(64)	92	W	(87)	123	q	(113)	169	F3	(243)
13	CR	(13)	65	A	(65)	93	X	(88)	124	r	(114)	170	F4	(244)
14	SO	(14)	65	À	(161)	94	Y	(89)	125	s	(115)	171	ID	(245)
15	SI	(15)	65	Â	(162)	94	Ÿ	(238)	125	ß	(222)	172	CL	(128)
16	DL	(16)	65	Ã	(208)	95	Z	(90)	126	š	(236)	173	IV	(129)
17	D1	(17)	65	Æ	(211)	96	þ	(240)	127	t	(116)	174	BG	(130)
18	D2	(18)	65	Ä	(216)	97	[	(91)	128	u	(117)	175	IB	(131)
19	D3	(19)	65	Á	(224)	98	\	(92)	128	û	(195)	176	UL	(132)
20	D4	(20)	65	Å	(225)	99	]	(93)	128	ú	(199)	177	IU	(133)
21	NK	(21)	66	B	(66)	100	^	(94)	128	ù	(203)	178	BL	(134)
22	SY	(22)	67	C	(67)	101	_	(95)	128	ü	(207)	179	IB	(135)
23	EB	(23)	67	Ç	(180)	102	`	(96)	129	v	(118)	180	WH	(136)
24	CN	(24)	69	D	(68)	103	a	(97)	130	w	(119)	181	RD	(137)
25	EM	(25)	70	Ð	(227)	103	â	(192)	131	x	(120)	182	YE	(138)
26	SB	(26)	71	E	(69)	103	á	(196)	132	y	(121)	183	GR	(139)
27	EC	(27)	71	È	(163)	103	à	(200)	132	ÿ	(239)	184	CY	(140)
28	FS	(28)	71	Ê	(164)	103	ä	(204)	133	z	(122)	185	BU	(141)
29	GS	(29)	71	Ë	(165)	103	å	(212)	134	þ	(241)	186	MG	(142)
30	RS	(30)	71	É	(220)	103	æ	(215)	135	{	(123)	187	BK	(143)
31	US	(31)	72	F	(70)	103	ã	(226)	136	|	(124)	188	90	(144)
32		(32)	73	G	(71)	104	b	(98)	137	}	(125)	189	91	(145)
33	!	(33)	74	H	(72)	105	c	(99)	138	~	(126)	190	92	(146)
34	"	(34)	75	I	(73)	105	ç	(181)	139	´	(168)	191	93	(147)
35	#	(35)	75	Î	(166)	107	d	(100)	140	`	(169)	192	94	(148)
36	$	(36)	75	Ï	(167)	108	đ	(228)	141	^	(170)	193	95	(149)
37	%	(37)	75	Í	(229)	109	e	(101)	142	¨	(171)	194	96	(150)
38	&	(38)	75	Ì	(230)	109	ê	(193)	143	~	(172)	195	97	(151)
39	'	(39)	76	J	(74)	109	é	(197)	144	£	(175)	196	98	(152)
40	(	(40)	77	K	(75)	109	è	(201)	145	¯	(176)	197	99	(153)
41	)	(41)	78	L	(76)	109	ë	(205)	146	·	(179)	198	9A	(154)
42	*	(42)	80	M	(77)	110	f	(102)	147	¡	(184)	199	9B	(155)
43	+	(43)	81	N	(78)	111	g	(103)	148	¿	(185)	200	9C	(156)
44	,	(44)	82	Ñ	(182)	112	h	(104)	149	¤	(186)	201	9D	(157)
45	-	(45)	83	O	(79)	113	i	(105)	150	£	(187)	202	9E	(158)
46	.	(46)	83	Ø	(210)	113	î	(209)	151	¥	(188)	203	9F	(159)
47	/	(47)	83	Õ	(218)	113	í	(213)	152	§	(189)	204	K	(255)
48	0	(48)	83	Ô	(223)	113	ì	(217)	153	ƒ	(190)			
49	1	(49)	83	Ó	(231)	113	ï	(221)	154	¢	(191)			
50	2	(50)	83	Ò	(232)	114	j	(106)	155	–	(246)			
51	3	(51)	83	Õ	(233)	115	k	(107)	156	¼	(247)			

SWEDISH Lexical Order

The SWEDISH lexical order table includes one "2 for 1" character replacement entry. When the "ß" character is found in a string, two sequence numbers are generated, as if two characters were found in the string.

ß = ss

Case Conversions

The following lists show the UPC$ and LWC$ transformations for the SWEDISH lexical order.

UPC$

```
abcdefghijklmnopqrstuvwxyzçñâêôûáéóúàèòùäëöüîàíøæìïßãđõšýÞ

ABCDEFGHIJKLMNOPQRSTUVWXYZÇÑÂÊÔÛÁÉÓÚÀÈÒÙÄËÖÜÎÀÍØÆÌÏßÃĐÕŠÝÞ
```

LWC$

```
ABCDEFGHIJKLMNOPQRSTUVWXYZÀÂÈÊÉÎÏÙÛÇÑÃØÆÄÖÜÉÔÁÃĐÍÌÓÒÕŠÚÝÞ

abcdefghijklmnopqrstuvwxyzàâèêéîïùûçñaøæäöüéôáãđíìóòõšúýþ
```

LEXICAL ORDER IS SWEDISH

Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.	Seq.	Chr.	Num.
0	NU	(0)	52	4	(52)	104	f	(229)	154	æ	(215)	206	1/2	(248)
1	SH	(1)	53	5	(53)	105	t	(230)	155	á	(212)	207	a	(249)
2	SX	(2)	54	6	(54)	106	f	(166)	156	á	(196)	208	o	(250)
3	EX	(3)	55	7	(55)	107	ï	(167)	157	à	(200)	209	«	(251)
4	ET	(4)	56	8	(56)	108	Ñ	(182)	158	â	(192)	210	■	(252)
5	EQ	(5)	57	9	(57)	109	ó	(231)	159	ä	(204)	211	»	(253)
6	AK	(6)	58	:	(58)	110	ò	(232)	160	ã	(226)	212	±	(254)
7	BL	(7)	59	;	(59)	111	ô	(223)	161	ç	(181)	213	▒	(127)
8	ES	(8)	60	<	(60)	112	ö	(218)	162	đ	(228)	214		(160)
9	HT	(9)	61	=	(61)	113	õ	(233)	163	è	(201)	215	B1	(177)
10	LF	(10)	62	>	(62)	114	Ø	(210)	164	ê	(193)	216	B2	(178)
11	VT	(11)	63	?	(63)	115	š	(235)	165	ë	(205)	217	F2	(242)
12	FF	(12)	64	@	(64)	116	ú	(237)	166	í	(213)	218	F3	(243)
13	CR	(13)	65	A	(65)	117	ù	(173)	167	ì	(217)	219	F4	(244)
14	SO	(14)	66	B	(66)	118	û	(174)	168	î	(209)	220	ID	(245)
15	SI	(15)	67	C	(67)	119	ü	(219)	169	ï	(221)	221	CL	(128)
16	DL	(16)	68	D	(68)	120	ÿ	(238)	170	ñ	(183)	222	IV	(129)
17	D1	(17)	69	E	(69)	121	þ	(240)	171	ó	(198)	223	BG	(130)
18	D2	(18)	70	F	(70)	122	[	(91)	172	ò	(202)	224	IB	(131)
19	D3	(19)	71	G	(71)	123	\	(92)	173	ô	(194)	225	UL	(132)
20	D4	(20)	72	H	(72)	124	]	(93)	174	ö	(206)	226	IU	(133)
21	NK	(21)	73	I	(73)	125	^	(94)	175	õ	(234)	227	BG	(134)
22	SY	(22)	74	J	(74)	126	_	(95)	176	ø	(214)	228	IB	(135)
23	EB	(23)	75	K	(75)	127	`	(96)	177	š	(236)	229	HH	(136)
24	CN	(24)	76	L	(76)	128	a	(97)	178	ú	(199)	230	RD	(137)
25	EM	(25)	77	M	(77)	129	b	(98)	179	ù	(203)	231	YE	(138)
26	SB	(26)	78	N	(78)	130	c	(99)	180	û	(195)	232	GR	(139)
27	EC	(27)	79	O	(79)	131	d	(100)	181	ü	(207)	233	CY	(140)
28	FS	(28)	80	P	(80)	132	e	(101)	182	ÿ	(239)	234	BU	(141)
29	GS	(29)	81	Q	(81)	132	é	(197)	183	þ	(241)	235	MG	(142)
30	RS	(30)	82	R	(82)	133	f	(102)	184	{	(123)	236	BK	(143)
31	US	(31)	83	S	(83)	134	g	(103)	185	|	(124)	237	90	(144)
32		(32)	84	T	(84)	135	h	(104)	186	}	(125)	238	91	(145)
33	!	(33)	85	U	(85)	136	i	(105)	187	~	(126)	239	92	(146)
34	"	(34)	86	V	(86)	137	j	(106)	188	´	(168)	240	93	(147)
35	#	(35)	87	W	(87)	138	k	(107)	189	`	(169)	241	94	(148)
36	$	(36)	88	X	(88)	139	l	(108)	190	^	(170)	242	95	(149)
37	%	(37)	89	Y	(89)	140	m	(109)	191	¨	(171)	243	96	(150)
38	&	(38)	90	Z	(90)	141	n	(110)	192	~	(172)	244	97	(151)
39	'	(39)	91	Æ	(211)	142	o	(111)	193	£	(175)	245	98	(152)
40	(	(40)	92	Ä	(208)	143	p	(112)	194	¯	(176)	246	99	(153)
41	)	(41)	93	Á	(224)	144	q	(113)	195	·	(179)	247	9A	(154)
42	*	(42)	94	À	(161)	145	r	(114)	196	¡	(184)	248	9B	(155)
43	+	(43)	95	Â	(162)	146	s	(115)	197	¿	(185)	249	9C	(156)
44	,	(44)	96	Ä	(216)	146	ß	(222)	198	¤	(186)	250	9D	(157)
45	-	(45)	97	Ã	(225)	147	t	(116)	199	£	(187)	251	9E	(158)
46	.	(46)	98	Ç	(180)	148	u	(117)	200	¥	(188)	252	9F	(159)
47	/	(47)	99	Ð	(227)	149	v	(118)	201	§	(189)	253	K	(255)
48	0	(48)	100	É	(220)	150	w	(119)	202	ƒ	(190)			
49	1	(49)	101	È	(163)	151	x	(120)	203	¢	(191)			
50	2	(50)	102	Ê	(164)	152	y	(121)	204	-	(246)			
51	3	(51)	103	Ë	(165)	153	z	(122)	205	1/4	(247)			

User-defined LEXICAL ORDER

The following program will generate the worksheet on the next page. The worksheet is handy when creating a user-defined lexical order.

```
10    DIM Lb$[1],F1$[23],F2$[23],F3$[14],F4$[20],Falt$[96],F1p$[22],F2p$[22]
20    INTEGER I
30    OUTPUT PRT;"LEXICAL ORDER TABLE WORKSHEET     !seq-num:mode-type,mode-entr
y!"
40    OUTPUT PRT
50    Lb$="#"
60    F1p$=" ,DD,X,""!    :  ,   !!"""
70    F1$=" ,DDD,X,""!    :  ,   !!"""
80    F2p$=",X,A,X,""!    :  ,   !!"""
90    F2$=",XX,A,X,""!    :  ,   !!"""
100   F4$=" ,DD,X,""!    :   !!"""
110   F3$=","Mode Length"""
120   Falt$=F1p$&F2$&F1$&F2$
130   FOR I=0 TO 63
140     SELECT I
150     CASE 0
160       OUTPUT PRT USING Lb$&Falt$&F3$;I,CHR$(I+64),I+128,CHR$(I+192)
170     CASE <32
180       OUTPUT PRT USING Lb$&Falt$&F4$;I,CHR$(I+64),I+128,CHR$(I+192),I-1
190     CASE ELSE
200       OUTPUT PRT USING Lb$&F2p$&RPT$(F2$,3)&F4$;CHR$(I),CHR$(I+64),CHR$(I+12
8),CHR$(I+192),I-1
210     END SELECT
220     OUTPUT PRT
230   NEXT I
240   END
```

LEXICAL ORDER TABLE WORKSHEET |seq-num:mode-type.mode-entry|

0	: .	@	: .	128	: .	â	: .	Mode	Length
1	: .	A	: .	129	: .	ê	: .	0	:
2	: .	B	: .	130	: .	ô	: .	1	:
3	: .	C	: .	131	: .	û	: .	2	:
4	: .	D	: .	132	: .	á	: .	3	:
5	: .	E	: .	133	: .	é	: .	4	:
6	: .	F	: .	134	: .	ó	: .	5	:
7	: .	G	: .	135	: .	ú	: .	6	:
8	: .	H	: .	136	: .	à	: .	7	:
9	: .	I	: .	137	: .	è	: .	8	:
10	: .	J	: .	138	: .	ò	: .	9	:
11	: .	K	: .	139	: .	ù	: .	10	:
12	: .	L	: .	140	: .	ä	: .	11	:
13	: .	M	: .	141	: .	ë	: .	12	:
14	: .	N	: .	142	: .	ö	: .	13	:
15	: .	O	: .	143	: .	ü	: .	14	:
16	: .	P	: .	144	: .	Å	: .	15	:
17	: .	Q	: .	145	: .	î	: .	16	:
18	: .	R	: .	146	: .	Ø	: .	17	:
19	: .	S	: .	147	: .	Æ	: .	18	:
20	: .	T	: .	148	: .	å	: .	19	:
21	: .	U	: .	149	: .	í	: .	20	:
22	: .	V	: .	150	: .	ø	: .	21	:
23	: .	W	: .	151	: .	æ	: .	22	:
24	: .	X	: .	152	: .	Ä	: .	23	:
25	: .	Y	: .	153	: .	ì	: .	24	:
26	: .	Z	: .	154	: .	Ö	: .	25	:
27	: .	[	: .	155	: .	Ü	: .	26	:
28	: .	\	: .	156	: .	É	: .	27	:
29	: .	]	: .	157	: .	ï	: .	28	:
30	: .	^	: .	158	: .	ß	: .	29	:
31	: .	_	: .	159	: .		: .	30	:
	: .	`	: .		: .		: .	31	:
!	: .	a	: .	Á	: .		: .	32	:
"	: .	b	: .	Í	: .		: .	33	:
#	: .	c	: .	Ó	: .		: .	34	:
$	: .	d	: .	Ú	: .		: .	35	:
%	: .	e	: .	Â	: .		: .	36	:
&	: .	f	: .	Ê	: .		: .	37	:
'	: .	g	: .	Ô	: .		: .	38	:
(	: .	h	: .	´	: .		: .	39	:
)	: .	i	: .	`	: .		: .	40	:
*	: .	j	: .	^	: .		: .	41	:
+	: .	k	: .	¨	: .		: .	42	:
,	: .	l	: .	~	: .		: .	43	:
-	: .	m	: .	Ê	: .		: .	44	:
.	: .	n	: .	Ô	: .		: .	45	:
/	: .	o	: .	₤	: .		: .	46	:
0	: .	p	: .	¯	: .		: .	47	:
1	: .	q	: .	Ā	: .		: .	48	:
2	: .	r	: .	ā	: .		: .	49	:
3	: .	s	: .	°	: .		: .	50	:
4	: .	t	: .	Ç	: .		: .	51	:
5	: .	u	: .	ç	: .		: .	52	:
6	: .	v	: .	Ñ	: .		: .	53	:
7	: .	w	: .	ñ	: .		: .	54	:
8	: .	x	: .	¡	: .		: .	55	:
9	: .	y	: .	¿	: .		: .	56	:
:	: .	z	: .	¤	: .		: .	57	:
;	: .	{	: .	£	: .		: .	58	:
<	: .			: .	º	: .		: .	59	:
=	: .	}	: .	§	: .		: .	60	:
>	: .	~	: .	Ω	: .		: .	61	:
?	: .	▒	: .	[illegible]	: .		: .	62	:

User-Defined Lexical Orders

A lexical order can be created for applications that require special collating sequences. If you can use one of the predefined lexical orders, you may wish to only skim this section.

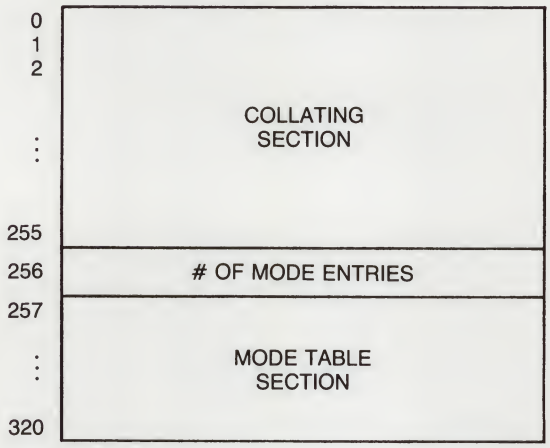
A program called LEX_AID has been supplied (on the BASIC Utilities Library disc) to simplify the creation of user-defined lexical orders. Before running the program it will be neccessary to have an understanding of the terms used in this section. Using the LEX_AID program is described in the BASIC Utilities Library manual.

Basically, a 321 element (0 thru 320) INTEGER array is dimensioned, filled with sequence numbers and mode entries, and the new lexical order is established by the following statement.

```
LEXICAL ORDER IS Table(*)
```

Where Table(*) is any valid INTEGER array name.

The following illustration shows the general construction of a user-defined lexical table created in an INTEGER array.



The first 256 elements (0 through 255) contain the sequence number to be used in place of the character's ASCII value. For special characters, a mode type and mode table pointer are also stored in these elements.

The next element (256) contains the number of entries in the mode table. This value can range from 0 (no mode table) thru 64 (a full mode table).

The remaining 64 elements (257 thru 320) contain the optional mode table entries assigned to special characters.

Sequence Numbers

Normally, comparing two strings results in the computer comparing the ASCII values of the characters. When the computer makes the string comparison "A"<"B", the ASCII value of "A" (65) is compared to the ASCII value of the letter "B" (66) resulting in the comparison: 65<66, which is true.

Now suppose that a new value (sequence number) could be assigned to each of the ASCII characters. We might wish to assign the letter "A" a sequence number greater than the sequence number assigned to the letter "B". If such an assignment were made, the comparison "A"<"B", would now be false.

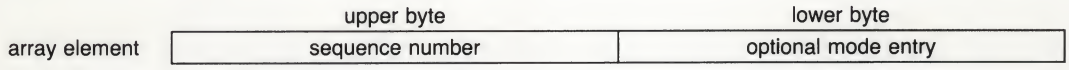
Once a lexical order is invoked, if two strings are compared, the strings are first converted into two series of sequence numbers and the comparison is then based on the sequence numbers.

The LEXICAL ORDER IS statement's primary purpose is to assign a sequence number to each character. However, this is not always enough to handle certain character combinations and special cases encountered in other languages. Special characters have a mode entry included with the sequence number.

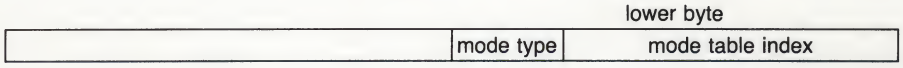
Mode Entries

Each of the first 256 array elements (0 thru 255) contains the sequence number to be used in place of the character's ASCII value. Optionally, a mode entry can be included.

Internally, an integer array element uses two bytes (16 bits) of memory. In the following diagram, the array element is divided into its upper, and lower bytes. The upper byte contains the sequence number and the lower byte is used if the character has a mode entry.



The lower byte is further divided into two parts. The upper-most 2-bits are used to represent one of the four mode types. The remaining 6-bits store an index (pointer) to the actual mode table entries. This method allows all the necessary information, for each character, to be stored as a single element in the INTEGER array.



Mode Type

Any one of the following mode types can be assigned to a character.

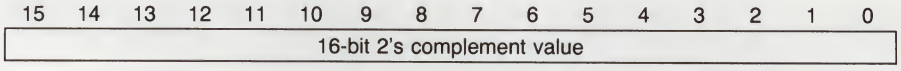
- Don't Care Characters (Mode type: 0)
- "1 for 2" Character Replacements (Mode type: 1)
- "2 for 1" Character Replacements (Mode type: 2)
- Accent Priority (Mode type: 3)

Mode Index
The mode index points to the actual mode table entry associated with the particular character. Up to 64 indexes are allowed (0 thru 63); however, some mode types use more than one table entry.

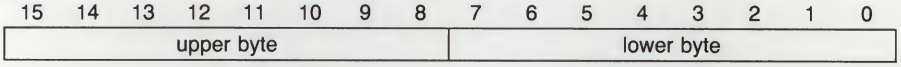
Bits, Bytes, and Mode Types

Each INTEGER array element stores a signed-integer in the range: −32768 thru 32767. Internally, the number is stored as a 16-bit 2's complement value.

Bits are usually numbered in descending order and include bit 0, so 16 bits are numbered as follows.



However, we want to store one of 256 possible sequence numbers and optionally, a mode type and mode table index. Since there are 256 characters used with the LEXICAL ORDER IS statement, and 8 bits are needed to store one of 256 possible values (2^8 = 256), it is convenient to think of the bits arranged as two bytes (a byte contains 8 bits).



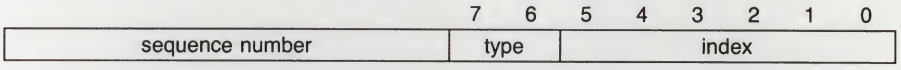
The upper byte is used to hold the sequence number and the lower byte contains the mode entry information. The algorithm below will produce a signed 16-bit integer from two unsigned 8-bit bytes.

```
Integer = (256*Upper + Lower) - (Upper>127)*65536
```

The process can be reversed.

```
IF Integer<0 THEN Integer=Integer+65536
Upper=Integer DIV 256
Lower=Integer MOD 256
```

The lower byte is further divided into two groups. Two bits hold one of four mode types (2^2=4) and the remaining six bits are for one of 64 mode indexes (2^6=64).



A "1 for 2" entry is signified by bit-6 being set. Therefore the value of the lower byte can range from 64 thru 126. (a "1 for 2" requires at least 2 entries.)

A "2 for 1" entry has bit-7 set. The value of the lower byte can range from 128 thru 191.

An "Accent priority" entry has both bit-6 and bit-7 set. The value of the lower byte ranges from 192 thru 255.

"Don't Care" Characters

A character can be removed from the collation sequence. To mark a character as a "don't care", the mode type is 0 (the same as a regular character) but the mode table index is set to 1.

sequence number	type	index
any value	0	1

The mode index need not point to a valid table entry, but must be a "1" to indicate a "don't care" character.

For example, the FRENCH lexical table lists the hyphen (-) as a "don't care" character. Thus, the hyphen is ignored when a string comparison is being made. The entry appears:

	sequence number	type	index
(45)	45	0	1

You may wish to include "don't care" characters in your own lexical tables. A string containing only "don't care" characters will match the null string.

The following short program illustrates the operation of a "don't care" character.

```
10 Return$="RESTORE"
20 Again$="RE-STORE"
30 !
40 LEXICAL ORDER IS ASCII
50 IF Restore$=Again$ THEN PRINT "True for ASCII"
60 LEXICAL ORDER IS FRENCH
70 IF Restore$=Again$ THEN PRINT "True for FRENCH"
80 END
```

Results:

```
True for FRENCH
```

"1 for 2" Character Replacement

This type of mode table entry indicates that one sequence number is to be used for two consecutive characters. It should be remembered that no characters are actually replaced by this operation, only that a single sequence number is to be used when the two characters are found adjacent to each other.

The following entry is placed in the collating section of the lexical table.

sequence number	type	mode index
normal sequence number	1	index

If a character marked as a "1 for 2" is found in a string, the next character is accessed and compared to the list of possible secondary characters in the mode table section.

(257 + index)	number of entries to check	
	second character	sequence number for this pair
	second character	sequence number for this pair

If the character does not match any of the secondary characters in the mode table, the original character's sequence number is used and processing continues. If a match is found, the sequence number for the pair is used and processing continues with the character following the secondary character.

For example, the SPANISH collating sequence has a "1 for 2" replacement for the letters "CH" or "Ch". The letter "C" is marked as a "1 for 2" character. When the letter is encountered in a string, the next character is accessed and compared to the list of possible secondary characters (uppercase H and lowercase h). The appropriate sequence number is then used for the pair. If the character following the letter "C" is not found in the list of possible secondary characters, the sequence number for "C" is used and processing continues with the next character.

You can override a "1 for 2" character replacement by inserting a "Don't Care" character between the two characters that would otherwise be replaced by a single sequence number.

The SPANISH table entry for the character sequence "CH" is below. In the collating section, the first letter of the sequence has the following entry:

	sequence number	type	mode index
(67)	67 (C)	1	1
(68)	69 (D)	0	0

(257 + 1)	number of entries to check (2)	
(257 + 2)	second character (H)	sequence number for pair (68)
(257 + 3)	second character (h)	sequence number for pair (68)

The sequence number assigned to the two-character combination is greater than the sequence number for the letter "C" and less than the sequence number for the letter "D". Therefore, a word beginning with the characters "CH" will collate after all words starting with the letter "C" followed by any other character.

The following program shows the sorting order for the letters "CH" in the SPANISH lexical order.

```
5 DIM A$(3)[3]
10 A$(1)="CGA"
20 A$(2)="CHA"
30 A$(3)="CIA"
40 LEXICAL ORDER IS SPANISH
50 MAT SORT A$(*)
60 PRINT A$(*)
70 END
```

Produces:

```
CGA CIA CHA
```

It should be noted that a character may have more than one secondary character combination. This is demonstrated by having both upper and lower case entries. Other secondary characters could have been included in the same manner. The first mode table entry contains the number of secondary characters to check and must be in the range: 0 thru 63.

"2 for 1" Character Replacement

When a "2 for 1" mode entry is specified, it indicates that the character should be represented by two sequence numbers (as if there were two characters in the string). The first sequence number is stored with the character as usual. The mode index points to the mode table entry that contains the second sequence number to be used for that character.

sequence number	type	mode index
1st sequence number	2	index

The mode table entry actually contains two sequence numbers. If the original character was upper case, the next character in the string will determine whether the upper or the lower sequence number is used. If the original character was a lower case letter, the lower sequence number is always used.

	upper	lower
(257 + index)	2nd sequence number (UPC)	2nd sequence number (LWC)

Several "2 for 1" characters are in the GERMAN lexical order. For instance, the character "Ä" is equivalent to "AE" and has the following entry in the collating section.

	sequence number	type	mode index
(216)	65 (A)	2	index

The index points to the following entry in the mode table.

	upper	lower
(257 + index)	75 (E)	124 (e)

In some cases, such as the character "ß", both upper and lower bytes contain the sequence number for the same character(s). This results in the same sequence numbers being generated regardless of the case of the next character.

Accent Priority

Accent Priority can be used as the final arbitrator of string comparisons. If you examine the lexical tables you will often find the same sequence number assigned to more than one character. Therefore, it is possible for two different strings to produce identical series of sequence numbers. The two strings will be considered equal unless at least one character, in each string, has been assigned different accent priorities.

Accent priority is established by assigning a value, in the range: 0 thru 63, to the character. Any character not already assigned a mode type may be assigned a priority. A priority of zero is assumed for all characters that haven't been assigned a priority.

sequence number	type	mode index
normal sequence number	3	priority

In the FRENCH lexical order, the characters: A, Á, and À have been assigned the same sequence number (64). Assume the characters were assigned the following priorities.

Character	Priority	
A	0	(default priority)
Á	1	
À	2	

The characters can now be distinguished from one another and will collate in the following order.

$$A < Á < À$$

When two strings are compared, each string is first converted into a series of sequence numbers. The comparison is then determined (in most cases) by the greater sequence numbers or the longer series of sequence numbers.

In the event both strings produce identical series of sequence numbers, the series of priorities are checked. The string containing the characters with the higher priority is the greater string.

User-Defined Functions and Subprograms

Chapter 6

Introduction

One of the most powerful constructs available in any language is the subprogram (a user-defined function is a special form of subprogram). A subprogram can do everything a main program can do except that it must be invoked or "called" before it is executed, whereas a main program is executed by pressing the RUN key. In a sense, pressing the RUN key is how you "call" a main program.

A subprogram has its own "context" or state as distinct from a main program and all other subprograms. This means that every subprogram has its own set of variables, its own softkey definitions, its own DATA blocks, and its own line labels. There are several benefits to be realized by taking advantage of subprograms:

- The subprogram allows the programmer to take advantage of the "Top-Down" method of designing programs. In this technique, the problem to be solved is broken up into a set of smaller and more easily solvable problems. These smaller problems can in turn be broken up into smaller problems yet, and so on. This technique has been shown to greatly improve the design, coding, and testing of programs, and will be discussed further at the end of the chapter.
- By separating all the details of performing the subtasks from the overall logic flow of the main program, the program is much easier to read from the subprogram calls. The programmer can see at a high level what he's trying to accomplish, rather than immediately getting lost in the details of each little sub-task.
- One of the most time-consuming parts of writing a program is debugging it, or forcing it to run correctly. The time consuming part of fixing bugs in a program is finding where the bug is in the first place. By using subprograms and testing each one independently of the others, it is easier to locate problems, and hence to fix them.
- Often, a programmer may want to perform the same task from several different areas of his program. For example, a set of readings may need to be taken from a voltmeter after each of four different input signals are fed through a circuit being tested. The same subprogram may be used to set up the voltmeter and take the readings, while different pieces of code would have to be used to set up the differing input conditions. Thus, subprograms can be used to economize on the overall size of the program.
- Finally, libraries of commonly used subprograms can be assembled for widespread use. Many different users doing diverse types of problems still may require some identical subprograms. For instance, an engineer may be using a subprogram to plot an array of data that he gathered from a spectrum analyzer, while the marketing person down the hall may be using the same subprogram to plot an array of data representing next year's sales forecast.

Some Startup Details

Location

A subprogram is located after the body of the main program, following the main program's END statement. (The END statement must be the last statement in the main program except for comments.) Subprograms may not be nested with other subprograms, but are physically delimited from each other with their heading statements (SUB or DEF) and ending statements (SUBEND or FNEND).

Naming

A subprogram has a name which may be up to fifteen characters long, just as with line labels and variable names. Here are some legal subprogram names:

Initialize
Read_dvm
Sort_2_d_array
Plot_data
(Katakana name)

Because up to 15 characters are allowed for naming subprograms, it is easy and convenient to name subprograms in such a way as to reflect the purpose for which the subprogram was written.

For Example

The following example shows a program which uses subprograms.

```
10   OPTION BASE 1
20   DIM Numbers(20)
30   CALL Build_array(Numbers(*),20)
40   CALL Sort_array(Numbers(*),20)
50   PRINT FNSum_array(Numbers(*),20)
60   END
70   SUB Build_array(X(*),N)
80   ! X(*) is the array to be defined
90   ! N tells how many elements are in the array
100  !   (1 is assumed to be the lower index)
110  FOR I=1 TO N
120    DISP "ELEMENT #";I;
130    INPUT "?",X(I)
140  NEXT I
150  SUBEND
160  SUB Sort_array(A(*),N)
170  ! A(*) is array to be sorted
180  ! N tells how many elements are in the array (1 is assumed
190  !  to be the lower bound)
200  ! Sort the array (elements 1-N) in increasing order
210  ! Algorithm used: Shell sort or Diminishing increment sort
220  ! Ref: Knuth, Donald E., The Art of Computer Programming,
230  !  Vol. 3 (Sorting and Searching), (Addison-Wesley 1973)
240  !  pp. 84-85
250  INTEGER T,S,H,I,J
260  REAL Temp
270  T=INT(LOG(N)/LOG(2))    ! # of diminshing increments
280  FOR S=T TO 1 STEP -1
290    H=2^(S-1)             ! ...16,8,4,2,1
300    FOR J=H+1 TO N
310      I=J-H
320      Temp=A(J)
330 Decide: IF Temp>=A(I) THEN Insert
```

```
340 Switch: A(I+H)=A(I)
350         I=I-H
360         IF I>=1 THEN Decide
370 Insert: A(I+H)=Temp
380       NEXT J
390     NEXT S
400     SUBEND
410     DEF FNSum_array(A(*),N)
420     ! Add A(1),...,A(N)
430     INTEGER I
440     REAL Array_total
450     FOR I=1 TO N
460       Array_total=Array_total+A(I)
470     NEXT I
480     RETURN Array_total
490     FNEND
```

Lines 10 through 60 are the main program. As you can see, it does nothing but call subprograms, which in turn do all the work. Line 70 is the header for the subprogram which asks the user to enter the values stored in his array. Notice that the main program has declared the array's name to be Numbers(*), but the subprogram uses the name X(*) to deal with the same array. The subprogram can name its variables whatever it wants without interfering with variables used outside the subprogram's context. The only variables that can be affected outside the subprogram's context are those passed through the parameter list (as shown here) or through COM (discussed later). In both cases, the matching between the subprogram and the outside world is done through the position of the variable(s) in the parameter list or COM block, not the actual name of the variable(s).

Starting at line 160 is the next subprogram which sorts the array into ascending order. The comments at the front of the subprogram serve to discuss the definition of the parameters used, and what effect the subprogram has on them. Also, the algorithm used is given, along with the proper reference material. It is an excellent idea to give a list of such pertinent details at the front of all subprograms. This makes debugging, modifying, optimizing, and re-using the subprogram much easier.

Starting at line 410 we see an example of a function subprogram. Functions are similar to SUB subprograms in concept. This particular example just adds the elements of the array together and returns the final value to the main program, which prints it.

The Difference Between a Function and a Subprogram

A SUB subprogram (as opposed to a function subprogram) is invoked explicitly using the CALL statement. A function subprogram is called implicitly by using the function name in an expression. It can be used in a numeric or string expression the same way a constant would be used, or it can be invoked from the keyboard. A function's purpose is to return a single value (either a real number or a string).

There are several functions that are built into the BASIC language which can be used to return values, such as SIN, SQR, EXP, etc.

```
Y=SIN(X)+Phase
Root1=(-B+SQR(B*B-4*A*C))/(2*A)
```

Communication

As mentioned earlier, there are two ways for a subprogram to communicate with the main program or with other subprograms: parameter lists, and COM (blank and labeled).

Parameter Lists

The formal parameter list is part of the subprogram's definition, just like the subprogram's name. The formal parameter list tells how many values may be passed to a subprogram, the types of those values (string, INTEGER, REAL, array, I/O path name), and the names the subprogram will use to refer to those values. The subprogram has the power to demand that the calling context match the types declared in the formal parameter list exactly — otherwise an error results. The calling context provides a pass parameter list which corresponds with the formal parameter list provided by the subprogram. The pass parameter list provides the values for those inputs required by the subprogram, and also provides the storage for the output values. It is perfectly legal for both the formal and pass parameter lists to be null, or nonexistent.

Here is a sample formal parameter list showing which types each parameter demands:

```
SUB Read_dvm(@Dvm,A(*),INTEGER Lower,Upper,Status$,Errflag)
```

`@Dvm` is an I/O path name which may refer to either an I/O device or a mass storage file. Its name here implies that it is a voltmeter, but it is perfectly legal to redirect I/O to a file just by using a different ASSIGN with @Dvm.

`A(*)` is a REAL array. Its size is declared by the calling context. Without MAT, there is no way to find the size of the array except through information supplied explicitly by the calling context; hence the parameters Lower and Upper.

`Lower` and `Upper` are declared here to be INTEGERs. Thus, when the calling program invokes this subprogram, it must supply either INTEGER variables or INTEGER expressions, or an error will occur.

`Status$` is a simple string which presumably could be used to return the status of the voltmeter to the main program. The length of the string is defined by the calling context.

`Errflag` is a REAL number. The declaration of the string Status$ has limited the scope of the INTEGER keyword which caused Lower and Upper to require INTEGER pass parameters.

There are two ways for the calling context to send values to a subprogram — pass by value, and pass by reference. Using pass by value, the calling context supplies a value and nothing more. Using pass by reference, the calling context actually gives the subprogram access to the calling context's value area. The distinction is that a subprogram can not alter the value of data in the calling context if the data is passed by value, while the subprogram **can** alter the value of data in the calling context if the data is passed by reference.

The subprogram has no control over whether its parameters are sent using pass by value or pass by reference. That is determined by the calling context's pass parameter list. In order for a parameter to be passed by reference, the pass parameter list (in the calling context) must use a variable for that parameter. In order for a parameter to be passed by value, the pass parameter list must use an expression for that parameter. Note that enclosing a variable in parentheses is sufficient to create an expression. Using pass by value, it is possible to pass an integer expression to a REAL formal parameter (the INTEGER is converted to its REAL representation) without causing a type mismatch error. Likewise, it is possible to pass a REAL expression to an INTEGER formal parameter (the value of the expression is rounded to the nearest INTEGER) without causing a type mismatch error (an integer overflow error is generated if the expression is out of range for an INTEGER). Let's look at our previous example from the calling side:

```
CALL Read_dvm(@Voltmeter,Readings(*),1,400,Status$,Errflag)
```

`@Voltmeter` is the pass parameter which matches the formal parameter @Dvm in the subprogram. I/O path names are always passed by reference, which means the subprogram can close the I/O path or assign it to a different file or device.

`Readings(*)` matches the array `A(*)` in the subprogram's formal parameter list. Arrays too, are always passed by reference.

1, 400 are the values passed to the formal parameters Lower and Upper. Since constants are classified as expressions rather than variables, these parameters have been passed by value. Thus, if the subprogram used either Lower or Upper on the left hand side of an assignment operator, no change would take place in the calling context's value area.

`Status$` is passed by reference here. If it were enclosed in parentheses, it would be passed by value. Notice that if it were passed by value, it would be totally useless as a method for returning the status of the voltmeter to the calling context.

`Errflag` is passed by reference.

OPTIONAL Parameters

Another important feature of formal parameter lists is the OPTIONAL keyword. Any formal parameter list (the one defining the subprogram) may contain the keyword OPTIONAL somewhere, although it isn't required to. The OPTIONAL keyword indicates that any parameters that follow it are not required in the pass parameter list of a calling context — they are optional. On the other hand, all parameters preceding the OPTIONAL keyword are required. If no OPTIONAL appears in the subprogram's parameter list, then all the parameters must be specified, or an error will be generated. The rules requiring matching of parameter types apply to OPTIONAL parameters as well as to ordinary parameters. There is a standard function called NPAR which can be used inside the subprogram to find out how many pass parameters the calling context actually did use. (NPAR will return 0 if used inside the main program, or if no parameters were passed to a subprogram.)

The OPTIONAL/NPAR combination is very effectively used in situations requiring external instrument setups. Most instruments have several different ranges, modes, settings, etc., which can be used depending upon the requirements of the user. Often, the user doesn't require the entire flexibility the instrument has to offer, and would rather use some reasonable defaults.

Consider the HP 3437A Digital Voltmeter. Among other things, this device has two data formats (packed and ASCII), three trigger modes (internal, external, and hold/manual), three voltage ranges (0.1V, 1V, and 10V), and also has programmable values for delay between readings, and numbers of readings taken. Naturally, the values used for the various settings will depend entirely upon the application for which the voltmeter is being used, but let's make some assumptions:

- The values for delay and number of readings are going to be changed frequently, so they will not be OPTIONAL parameters.
- Of the remaining OPTIONAL parameters, the range is most likely to be altered.

A reasonable setup routine for the voltmeter might look like this:

```
2010  SUB Setup_dvm(@Dvm,INTEGER Readings,REAL Delay,OPTIONAL INTEGER Prange,Ptr
igger,Pformat)
2020  SELECT NPAR
2030  CASE 3
2040    Format=1                              ! Default ASCII format
2050    Trigger=1                             ! Default internal trigger
2060    Range=2                               ! Default 1 volt range
2070  CASE 4
2080    Format=1
2090    Trigger=1
2100    Range=Prange
2110  CASE 5
2120    Format=1
2130    Trigger=Ptrigger
2140    Range=Prange
2150  CASE 6
2160    Format=Pformat
2170    Trigger=Ptrigger
2180    Range=Prange
2190  END SELECT
2200  OUTPUT @Dvm;"N";VAL$(Readings);"SD";VAL$(Delay);"SR";VAL$(Range);"T";VAL$(
Trigger);"F";VAL$(Format)
2210  SUBEND
```

Legal invocations of the Setup_dvm subprogram are:

```
570   Setup_dvm(@Dvm,100,,.001)       ! Default Range,Trigger,Format
630   Setup_dvm(@Dvm,500,,.05,3)      ! Default Trigger,Format
850   Setup_dvm(@Dvm,50,,.005,1,2)    ! Default Format
1010  Setup_dvm(@Dvm,70,,.075,2,1,2) ! Explicitly declare all values
```

Notice in the example above that local variables are used instead of the formal parameters. This is because it is illegal to use an OPTIONAL parameter variable if that variable was not passed from the calling context.

Other applications of the OPTIONAL/NPAR feature are limited only by the imagination, but here are a few ideas:

Write a subprogram which sorts an array in ascending order unless an OPTIONAL parameter tells it to sort in descending order.

Write a rootfinder routine which has an acceptance tolerance of $\pm 10^{-6}$ unless overridden with an OPTIONAL parameter.

Write a program which keeps track of departmental expenses, including the account billed, the item or service purchased, the person incurring the expense, and optionally, the person authorizing the expense.

COM Blocks

Since we've discussed parameter lists in detail, let's turn now to the other method a subprogram has of communicating with the main program or with other subprograms, the COM block.

There are two types of COM (or common) blocks, blank and labeled. Blank COM is simply a special case of labeled COM (it is the COM whose name is nothing) with the exception that blank COM must be declared in the main program, while labeled COM blocks don't have to be declared in the main program. Both types of COM blocks simply declare blocks of data which are accessible to any context having matching COM declarations.

A blank COM block might look like this:

```
10 OPTION BASE 1
20 COM Conditions(15),INTEGER,Cmin,Cmax,@Nuclear_pile,
   Pile_status$[20],Tolerance
```

A labeled COM might look like this:

```
30 COM /Valve/ Main(10),Subvalves(10,15),@Valve_ctrl
```

A COM block's name, if it has one, will immediately follow the COM keyword, and will be set off with slashes, as shown above. The same rules used for naming variables and subprograms are used for naming COM blocks.

Any context need only declare those COM blocks which it needs to have access to. If there are 150 variables declared in 10 COM blocks, it isn't necessary for every context to declare the entire set — only those blocks that are necessary to each context need to be declared. COM blocks with matching names must have matching definitions. As in parameter lists, matching COM blocks is done by position and type, not by name.

There are several characteristics of COM blocks which distinguish them from parameter lists as a means of communications between contexts.

- COM survives pre-run — In general, any numeric variable is set to 0, strings are set to the null string, and I/O path names are set to undefined after pushing the RUN key, or upon entering a subprogram. This is true of COM the first time the RUN key is pressed, but after COM block variables are defined, they retain their values until:
  1. SCRATCH A or SCRATCH C is executed,
  2. A statement declaring a COM block is modified by the user,
  3. A new program is brought into memory using the GET or LOAD commands which doesn't match the declaration of a given COM block, or which doesn't declare a given COM block at all.
- COM blocks can be arbitrarily large — One limitation on parameter lists (both pass and formal parameter lists) is that they must fit into a single program line along with the line's number, possibly a label, the invocation or subprogram header, and possibly (in the case of

a function) a string or numeric expression. Depending upon the situation, this can impose a restriction on the size of your parameter lists.

COM blocks can take as many statements as necessary. COM statements can be interwoven with other statements (though this is considered a poor practice). All COM statements within a context which have the same name will be part of the definition of that COM block.

- COM blocks can be used for communicating between contexts that do not invoke each other — Information such as modes and states can be an integral part of communicating between contexts, even though those contexts don't explicitly call each other. For instance, one routine might be responsible for setting the voltage range on a voltmeter, while another routine which may need to know what the current voltage range is in order to set up the scale on a graph properly.
- COM blocks can be used to communicate between subprograms that are not in memory simultaneously — Similar to the case above, subprograms can communicate with each other through COM blocks even though combinations of LOADSUB/DELSUB may preclude their simultaneous presence in memory.
- COM blocks can be used to retain the value of "local" variables between subprogram calls — In general, the variables used by a subprogram are discarded when the subprogram is exited. However, there are situations where it might be useful for a subprogram to "remember" a value. A machine which tests capacitors in an incoming inspection department may require calibration after every 100 tests are performed. If the subprogram which does the testing has a way to count how many tests it has already performed (using a labeled COM block), then this task can be left to the testing routine, simplifying the rest of the system.
- COM blocks allow subprograms to share data without the intervention of the main program — Subprogram libraries may consist of elaborate relationships of both programs and data structures. In many cases, a major portion of the data structures are only used for support of the task being performed, rather than being integral to the task itself. Thus the main program does not need to declare the supportive data structures.

  Examples of this situation might include data base management libraries (hashing tables may need to be maintained for accessing data quickly) or three dimensional graphics libraries (window, viewport, and clip information need to be kept, as well as object definitions and related transformations).

Hints for Using COM Blocks

Any COM blocks needed by your program must be resident in memory at prerun time (prerun is caused by pressing RUN, executing a RUN command, executing LOAD or GET from the progam, or executing a LOAD or GET from the keyboard and specifying a run line.) Thus if you want to create libraries of subprograms which share their own labeled COM blocks, it is wise to collect all the COM declarations together in one subprogram to make it easy to append them to the rest of the program for inclusion at prerun time. (The subprogram need not contain anything but the COM declarations.)

COM can be used to communicate between programs which overlay each other using LOAD or GET statements, if you remember a few rules.

1. COM blocks which match each other exactly between the two programs will be preserved intact. "Matching" requires that the COM blocks are named identically (except blank COM), and that corresponding blocks have exactly the same number of variables declared, and that the types and sizes of these variables match.
2. Any COM blocks existing in the old program which are not declared in the new program (the one being brought in with the LOAD or GET) are destroyed.
3. Any COM blocks which are named identically, but which do not match variables and types identically, are defined to match the definition of the new program. All values stored in that COM block under the old program are destroyed.
4. Any new COM blocks declared by the new program (including those mentioned above in #3 are initialized implicitly. Numeric variables and arrays are set to zero, strings are set to the null string, and I/O path names are set to undefined.

The first occurrence in memory of a COM block is used to define or set up the block. Subsequent occurrences of the COM block must match the defining block, both in the number of items, and the types of the items. In the case of strings and arrays, the actual sizes need be specified only in the defining COM blocks. Subsequent occurrences of the COM blocks may either explicitly match the size specifications by re-declaring the same size, or they may implicitly match the size specifications. In the case of strings, this is done by not declaring any size, just declaring the string name. In the case of arrays, this is done by using the `(*)` specifier for the dimensions of the array instead of explicitly re-declaring the dimensions.

Consider the following COM block definition:

```
10    COM /Dvm_state/ INTEGER Range,Format,N,REAL
      Delay,Lastdata(1:40),Status$[20]
```

The following occurrence of the same COM block within a subprogram matches the COM block explicitly and is legal:

```
2000 COM /Dvm_state/ INTEGER Range,Format,N,REAL
       Delay,Lastdata(1:40),Status$[20]
```

The following block within a different subprogram uses implicit matching and is also legal:

```
4010 COM /Dvm_state/ INTEGER Range,Format,N,REAL
       Delay,Lastdata(*),Status$
```

The following declaration is illegal, since it uses explicit size specifications on the array and string which do not match the original definition from line 10.

```
5020 COM /Dvm_state/ INTEGER Range,Format,N,REAL
       Delay,Lastdata(1:30),Status$[15]
```

The following declaration is also illegal, since it violates the types set forth by the defining block.

```
6010 COM /Dvm_state/ Range,Format,N,REAL
     Delay,Lastdata(*),Status$
```

In general, the implicit size matching on arrays and strings is preferable to the explicit matching because it makes programs easier to modify. If it becomes necessary to change the size of an array or string in a COM block, it only needs to be changed in one statement, the one which defines the COM block. If all other occurrences of the COM block use the (*) specifier for arrays, and omit the length field in strings, none of those statements will have to be changed as a result of changing an array or string size.

Context Switching

As mentioned in the introduction to this chapter, a subprogram has its own **context** or state as distinct from a main program and all other subprograms. In between the time that a CALL statement is executed (or an FN name is used) and the time that the first statement in the subprogram gets executed, the computer performs a "prerun" on the subprogram. This "entry" phase is what defines the context of the subprogram. The actions performed at subprogram entry are similar, but not identical, to the actual prerun performed at the beginning of a program. Here is a summary:

- The calling context has a DATA pointer which points to the next item in the current DATA block which will be used the next time a READ is executed (assuming of course that a DATA block even exists in the calling program). This pointer is saved whenever a subprogram is called, and then the DATA pointer is reset to the first DATA statement in the new subprogram context.
- The RETURN stack for any GOSUBs in the current context is saved and set to the empty stack in the new context.
- The system priority of the current context is saved, and the called subprogram inherits this value. Any change to the system priority which takes place within the subprogram (or any of the subprograms which it calls in turn) is purely local, since the system priority is restored to its original value upon subprogram exit. This is an important consideration: If the subprogram is called as a result of an event-initiated GOSUB/CALL statement, any ON <event> GOTO/GOSUB/CALL/RECOVER condition set up in the called subprogram must have a higher priority assigned to it than the event responsible for the subprogram's invocation. Otherwise, the event is guaranteed **not** to cause an end of line branch.
- Any event-initiated GOTO/GOSUB statements are disabled for the duration of the subprogram. If any of the specified events occur, this will be logged, but no action will be taken. (The fact that an event did occur will be logged, but only once — multiple occurrences of the same event will not be serviced.) Upon exiting the subprogram, these event-initiated conditions will be restored to active status, and if any of these events occurred while the subprogram was being executed, the proper branches will be taken.
- Any event-initiated CALL/RECOVER statements are saved upon entering a subprogram, but the subprogram still inherits these ON conditions since CALL/RECOVER are global in scope. However, it is legal for the subprogram to redefine these conditions, in which case the original definitions are restored upon subprogram exit.
- The current value of OPTION BASE is saved, and the value for the subprogram (0 or 1, explicitly declared or defaulted) is used.
- The current DEG or RAD mode for trigonometric operations and graphics rotations is saved. The subprogram will inherit the current DEG or RAD setting, but if it gets changed within the subprogram, the original setting will be restored when the subprogram is exited.

Variable Initialization

Space for all arrays and variables declared is set aside, whether they are declared explicitly with DIM, REAL, or INTEGER, or implicitly just by using the variable. The entire value area is initialized as part of the subprogram's prerun. All numeric values are set to zero, all strings are set to the null string, and all I/O path names are set to undefined.

Subprograms and Softkeys

ON KEYs are a special case of the event-initiated conditions that are part of context switching. They are special because they are the only <event> conditions which give visible evidence of their existence to the user through the softkeys labels at the bottom of the CRT. These key labels are saved just as the event conditions are, and the labels get restored to their original state when the subprogram is exited, regardless of any changes the subprogram made in the softkey definitions. This means the programmer doesn't have to make any special allowances for re-enabling his keys and their associated labels after calling a subprogram which changes them — the language system handles this automatically.

It is important to remember that the called subprogram inherits the softkey labels. All the keys are still active in some sense; ON KEY...CALL/RECOVER will cause their original program branches to take place immediately if the proper key is pressed, and ON KEY...GOTO/GOSUB will log the fact that a key is pressed until the subprogram is exited, at which time the proper branch will occur. This latter case may cause some consternation on the part of the user if he presses a softkey expecting immediate action and nothing happens since the key was temporarily disabled due to a called subprogram. If the called subprogram is expected to take a noticeably long time to execute, it might be a good idea to explicitly remove the labels from the disabled softkeys using the OFF KEY statement. Thus, the user won't expect anything to happen as a result of pressing a softkey. This technique is also useful for guaranteeing that a given subprogram is **not** interrupted prematurely. (The DISABLE statement is useful for preventing program branches as a result of an event-initiated happening, although it will not turn off the softkey labels.)

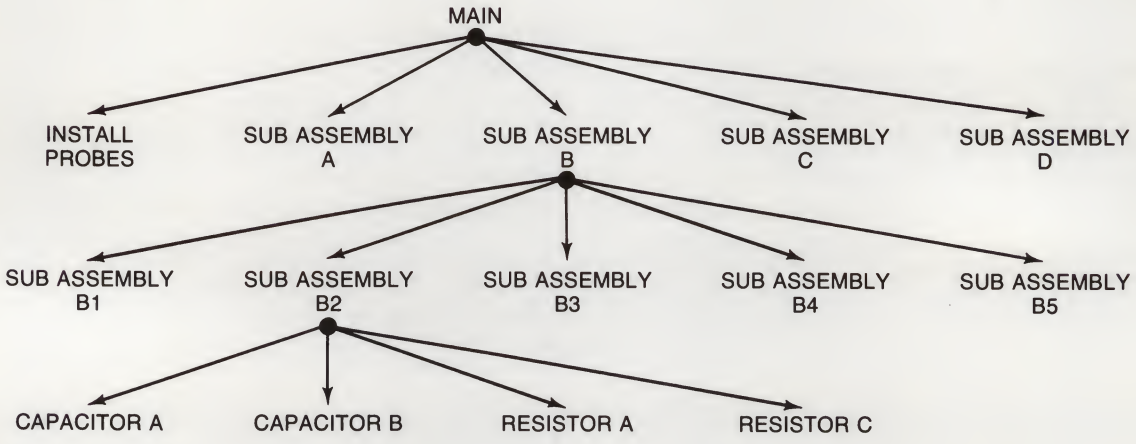
Subprograms and the RECOVER Statement

The event-initiated RECOVER statement allows the programmer to cause the program to resume execution at any given place in the context defining the ON...RECOVER as a result of a specified event occurring, regardless of subprogram nesting.

Thus, if a main program executes an ON...RECOVER statement (for example a softkey or an external interrupt from the SRQ line on an HP-IB), and then calls a subprogram, which calls a subprogram, which calls a subprogram, etc., program execution can be caused to immediately resume within the main program as a result of the specified event happening.

By way of illustration, consider the following example:

Suppose you are performing an exhaustive component test on a circuit board. The program may be designed like so:



When lunch break comes around, you may want to halt the current test so you can use the computer to play chess, or your boss might wander by and want to see the results of the rest of the tests performed this week. In either case, if the test program is nested three or four levels deep in subprograms, it might take a while for the test to complete. By defining a softkey to RECOVER to the main program, you can instantly terminate the test at any time, and make the computer available for something else. The RECOVER will discard anything being done in any of the subprograms between the context declaring the event-initiated RECOVER, and the subprogam being executed when the specified event occurs.

Again, the DISABLE statement can be used within any subprograms in which it is critical not to allow interruptions.

Live Keyboard

Functions and subprograms can be called from live keyboard by the user. There are some restrictions:

- Since variables cannot be created by the user from the keyboard (variables can only be defined by the program), it is legal to use only parameters that already exist in the current context.
- Constants may be used in the pass parameter list.
- When calling a SUB subprogram from the keyboard, the CALL keyword is not optional.

Speed Considerations

In some programs, speed is of the essence. In these cases, programmers will be reluctant to incur any unnecessary overhead in executing their task. There is a certain amount of overhead incurred in calling subprograms, although the overhead is fairly small, and shouldn't be an impediment to the use of subprograms. ("Overhead" is loosely defined to be the time it takes to perform those activities which aren't explicitly asked for by the user's program, but which are still necessary to keep the user's program running in a correct manner. The tasks discussed earlier under context switching are an excellent example of such overhead.)

Let's look at how much time it takes just to get in and out of the subprogram regardless of the task being performed by the subprogram. (The times in this discussion are approximate and apply to the Model 226.)

The time it takes to enter a subprogram depends upon the number of parameters being passed, the types of parameters being passed, and the number of variables declared local to the subprogram itself. To get in and out of a subprogram which has no parameters and which does nothing (in other words, a SUB followed by a SUBEND) takes 572 microseconds, meaning if you call it 1748 times, you'll lose about a second. (By way of comparison, 572 microseconds is about what it takes to perform four floating point additions. To perform four floating point additions and store the result from each one in a variable will take about 1080 microseconds, or just over a millisecond.)

Entry conditions	Approximate execution speed[1]
No parameters	572 μsec.
1 simple numeric	+ 105 μsec.
1 simple string	+ 128 μsec.
1 numeric array	+ 141 μsec.
1 string array	+ 141 μsec.
1 I/O path name	+ 123 μsec.
OPTION BASE in sub	+ 31 μsec.
REAL or INTEGER in sub	+ 32 μsec.
1st numeric array declaration	+ 18 μsec.
other numeric array declarations	+ 11 μsec.
1st string array declaration	+ 21 μsec.
other string array declarations	+ 12 μsec.

As you can see from the table, subprograms are a bargain in terms of speed. The relatively small amount of overhead required for invoking a subprogram is more than made up for by the benefits to be derived.

1 These speeds apply to computers without an HP 98635A, floating point.

Using Subprogram Libraries

If you have a program which is quite large, along with sizable data arrays, you could run out of memory in your computer. But the program you're working on just **has** to remain one program, and external factors prevent your reducing data array size. What to do? There are several options which address this problem.

If you want to load a specific subprogram from a PROG file, you would use the `LOADSUB <subprogram name> FROM` statement. If you want to load all the subprograms from a specific PROG file, you would use the `LOADSUB ALL FROM` statement. And, if you wanted to see which subprograms are still missing or load all those still needed, you would use the `LOADSUB FROM` command. Note that this is a *command*, and not a statement. Therefore `LOADSUB FROM` cannot be invoked programmatically.

Loading Subprograms One at a Time

Suppose your program has several options to select from, and each one needs many subprograms and much data. All the options, however, are mutually exclusive; that is, whichever option you choose, it does not need anything that the other options use. This means that you can clean up everything you've used when you are through with that option.

If all of your subprograms can be put onto one file, you can selectively retrieve them as needed with this sort of statement:

```
LOADSUB Subprog_1 FROM "SUBFILE"
LOADSUB Subprog_2 FROM "SUBFILE"
LOADSUB FNNumeric_fn FROM "SUBFILE"
LOADSUB FNString_function$ FROM "SUBFILE"
```

Note that only one subprogram per line can be loaded with this form of LOADSUB. If, for any program option, you need so many subprograms that this method would be cumbersome, you could use the following form of the command.

Loading Several Subprograms at Once

For this method, you store **all** the subprograms needed for each option on its own file. Then, when the program's user selects Program Option 1, you could have this line of code execute:

```
LOADSUB ALL FROM "OPT1SUBFL"
```

and if the user selects Option 2,

```
LOADSUB ALL FROM "OPT2SUBFL"
```

and so forth.

There is one other form of LOADSUB, but it cannot be used programmatically. This is covered next.

Loading Subprograms Prior to Execution

In the *LOADSUB FROM* form, for which you need PDEV, neither `ALL` nor a subprogram name is specified in the command. This is used prior to program execution. It looks through the program in memory, notes which subprograms are needed (referenced) but not loaded, goes to the specified file and attempts to load all such subprograms. If the subprograms are found on the file, they are loaded into memory; if they are not, an error message is displayed and a list of the subprograms still needed but not found in the file is printed.

This can be handy in two ways. The first and obvious way is that subprograms can be loaded quickly. The other way is this: suppose that you are developing a program and as you are coding, you realize you need a subprogram that does such-and-such. But your train of thought is chugging along so smoothly, you do not want to interrupt your coding of the routine you are working on to do the other little subprogram. But when the big one is done, you have forgotten all about coding the little one. If you suspect you've done this, the LOADSUB FROM command is very useful. Type a LOADSUB FROM command where the file name is a file on which you **know** there are none of the subprograms you need (perhaps a null PROG file). Of course, no subprograms will be loaded, but *a list of those yet undefined will be printed.* These are the ones you still need to code. Naturally, if you have already coded them and stored them somewhere, go get them. But if you haven't, this is a simple way of listing those still to be entered.

Any COM blocks declared in subprograms brought into memory with a LOADSUB by a running program must already have been declared. LOADSUB does not allow new COM blocks to be added to the ones already in memory. Furthermore, any COM blocks in the subprograms brought in must match a COM block in memory in both the number and type of the variables. Otherwise, an error occurs.

Note

If a main program is in a file referenced by a LOADSUB, it will **not be loaded**; only subprograms can be loaded with LOADSUB. Main programs are loaded with the LOAD command.

With all this talk of loading subprograms from files, one question arises: How do you get the subprograms **on** the file? Easily: type in the subprograms you want to be on one file, and then STORE them onto the desired file name. You must use STORE and not SAVE, because the LOADSUB looks for a PROG-type file. If you can't type in your subprograms error-free the first time (and who can?), what you can do is this: type in your program with all the subprograms it needs, and debug them. **After storing *everything* on a file for safekeeping,** delete what you do **not** want on the file, and STORE everything else on the subprogram file from which you will later do a LOADSUB. In this way, you know the subprograms will work when you load them.

Deleting Subprograms Programmatically

The utility of the LOADSUB commands would be greatly reduced if one could not delete subprograms from memory at will. So, there is a way to delete subprograms during execution of a program: DELSUB. If you want to delete only selected ones, you could type something like:

```
DELSUB Sort_data,Print_report_1,FNPoly_solve
```

If you are sure of the positioning of the subprograms in memory, here is a method of deleting whole groups of subprograms:

```
DELSUB Print_report TO END
```

You can combine these methods:

```
DELSUB Sort_data,Print_report,FNGet_name$ TO END
```

The subprograms to be deleted do not have to be contiguous in memory, nor does the order in which you specify the subprograms in a DELSUB statement have to be the order in which they occur in memory. The computer deletes each subprogram before moving on to the next name.

If there are any comments after a FNEND or SUBEND, but before the next SUB or DEF FN, these will be deleted as well as the rest of the subprogram body.

If the computer attempts to delete a nonexistent subprogram, an error occurs, and the DELSUB statement is terminated. This means that subprograms whose names occur after the error-causing one will not be deleted.

A subprogram can be deleted only if it is not currently active and if it is not referenced by a currently active ON RECOVER/CALL statement. This means:

1. A subprogram cannot delete itself.
2. A subprogram cannot delete a subprogram that called it, either directly or indirectly. (Otherwise it wouldn't have anywhere to return to when it finished!)

Between the time that a subprogram is entered and the time it is exited, the computer keeps track of an *activation record* for that subprogram. Thus, if a subprogram calls a subprogram that calls a subprogram, etc., none of the subsequently-called subprograms can delete the original one or any of the ones in between because the system knows from the activation record that control will eventually need to return to the original calling context. A similar situation exists with active event-initiated CALL/RECOVER statements. As long as the possibility of the specified event occurring exists, the system will not let the subprogram be deleted. In essence, the system will not let you execute two mutually-exclusive contradictory commands simultaneously.

Editing Subprograms

Inserting Subprograms

There are some rules to remember when inserting SUB and DEF FN statements:

It is not possible to insert a DEF FN or SUB statement in the middle of the program. All DEF FN and SUB statements must be appended to the **end** of the program. If you want to insert a subprogram in the middle of your program because your prefer to see it listed in a given order, you must perform the following sequence:

1. STORE the program.
2. Delete all lines **above** the point where you want to insert your subprogram (refer to the DEL statement).
3. STORE the remaining segment of the program in a new file.
4. LOAD the original program stored in step 1.
5. Delete all lines **below** the point where you want to insert your subprogram.
6. Type in the new subprogram.
7. Do a LOADSUB ALL from the new file in step 3.

With PDEV, the job is much easier:

1. Write your new subprogram *at the end* of the program.
2. Perform a MOVELINES command where:
    a. the Starting Line in the MOVELINES command is the line which you want to immediately follow your new subprogram,
    b. the Ending Line in the MOVELINES command is the line immediately prior to the SUB or DEF FN of the new subprogram, and
    c. The destination line is any line number greater than the highest line number currently in memory.

In either case there is an optional final step. It is not *required* that you do a REN to renumber the program at this point, but often it is desirable to close up the void left in the program line numbering which resulted from the block of subprograms being moved to the end of memory.

Deleting Subprograms
It is not possible to delete either DEF FN or SUB statements with the delete line key unless you first delete all the other lines in the subprogram. This includes any comments after the SUBEND or FNEND. Another way to delete DEF FN and SUB statements is to delete the entire subprogram, up to, but **not** including, the next SUB or DEF FN line (if any). This can be done either with the DEL command, or with the DELSUB command.

Merging Subprograms
If you want to merge two subprograms together, first examine the two subprograms carefully to insure that you don't introduce conflicts with variable usage and logic flow. If you've convinced yourself that merging the two subprograms is really necessary, here's how you go about it:

1. SAVE everything in your program **after** the SUB or DEF statement you want to delete.
2. Delete everything in your program from the unwanted SUB statement to the end.
3. GET the program segment you saved away in step 1 back into memory, taking care to number the segment in such a way as not to overlay the part of the program already in memory.

Once again, with PDEV, your job is greatly simplified:

Execute a MOVELINES command in which you move everything from one subprogram—**excluding the SUB/DEF FN and SUBEND/FNEND statements**—into the desired position in the other subprogram. If there are any declarative statements in the moved code, you will probably want to move those up next to the declarative statements in the receiving code. Don't forget to go back to the place where the code came from and delete the SUB/DEF FN statement and the SUBEND/FNEND statements.

SUBEND and FNEND

The SUBEND and FNEND statements must be the last statements in a SUB or function subprogram, respectively. These statements don't ever have to be executed; SUBEXIT and RETURN are sufficient for exiting the subprogram. (If SUBEND is executed, it will behave like a SUBEXIT. If FNEND is executed, it will cause an error.) Rather, SUBEND and FNEND are delimiter statements that indicate to the language system the boundaries between subprograms. The only exception to this rule is the comment statements (either REM or !), which are allowed after SUBEND and FNEND.

Notice that we haven't worried about the tree structure yet. This is because the tree is only used to provide ordering information to be used in printing out the report. Since the tree is not necessary except for the report, we'll let the report subprogram worry about it. The variables Max and Howmany are introduced for the sake of flexibility. It is possible that the department may have to hire more people at some time in the future, or that some people may leave the company or accept jobs in other departments. In this case, the program will have to be changed to allow for a different number of people. By making the maximum number of people, and the actual number of people, variables instead of constants, modifying the program becomes very easy.

Also, notice that each of the subprograms has been "stubbed in". The reason for doing this is that you can immediately run the program to test the communications between modules. So far, the program will not do anything, but it does allow you to make sure that your pass parameter lists match the formal parameter lists in the number and types of parameters. Furthermore, this process can be repeated every step of the way. As each subprogram is designed, the modules called by it can be "stubbed in" in a similar fashion, insuring that the parameter lists and communications paths are well defined and properly implemented at every level of your design. The most difficult part of testing your program is done as the program is being designed.

Let's step down to the next level of the design and consider each of the subprograms mentioned above:

```
90    SUB Input_data(Name$(*),INTEGER Units(*),REAL Failures(*),INTEGER Max,Howma
ny)
91    DIM Which$[3]
92    INPUT "New Data or Old?",Which$
93    IF Which$="New" THEN
94       Enter_new(Name$(*),Units(*),Failures(*),Max,Howmany)
95    ELSE
96       Edit_old(Name$(*),Units(*),Failures(*),Max,Howmany)
97    END IF
100   SUBEND
101   !
110   SUB Store_data(Name$(*),INTEGER Units(*),REAL Failures(*),INTEGER Max,Howma
ny)
111   Setup_file(@File)
112   OUTPUT @File;Name$(*),Units(*),Failures(*)
113   ASSIGN @File TO *
120   SUBEND
121   !
130   SUB Report(Name$(*),INTEGER Units(*),REAL Failures(*),INTEGER Max,Howmany)
132   OPTION BASE 1
133   INTEGER Root,I,Whichfield
134   ALLOCATE INTEGER Tree(Howmany,2)
135   Init_tree(Root,Tree(*))
136 Ask:  INPUT "Which field (1=Name,2=Units,3=Failures)?",Whichfield
137   IF Whichfield<1 OR Whichfield>3 THEN Ask
138   FOR I=2 TO Howmany
139      SELECT Which_field
140      CASE 1
141         Buildstring(Root,Tree(*),I,Name$(*))
142      CASE 2
143         Buildnum(Root,Tree(*),I,Units(*))
144      CASE 3
145         Buildnum(Root,Tree(*),I,Failures(*))
146      END SELECT
148   NEXT I
149   Inorder(Root,Tree(*),Name$(*),Units(*),Failures(*))
150   SUBEND
```

Here we haven't gone through the exercise of providing the dummy subprograms, though in actual practice we would. In lines 94 and 96 of the data entry program, we see two more subprograms that need to be designed. The module for entering new data from the keyboard will be straightforward and need not be considered in further detail for this example. The module for editing old data will involve loading a set of data from the diskette and then allowing the user to modify those values. This will involve a little more detail and perhaps another level of subprograms, but the techniques to be used are still straightforward enough not to demand further attention here.

Line 111 calls for a module to setup a data file to store the data on, and passes an I/O path name back out that's ready for use. This means that the subprogram must:

1. Ask the user for a file
2. Create the file if necessary
3. ASSIGN it for use

The Report subprogram is by far the most interesting one in this example, since it deals with the initialization, construction, and traversal of a binary tree, as discussed above. The Init_tree subprogram called in line 135 simply initializes the root node's (first element, remember) subtrees to be empty. Subsequently, the Buildstring subprogram called in line 140 simply enters the Ith string in the `Name$(*)` array into the structure of `Tree(*)`, assuming that the user asked for the report to be sorted by `Name$(*)`. Similarly, if the user wanted either `Units(*)` or `Failures(*)` to be the sort key, then the Buildnum subprogram (called in lines 143 and 145) would be used to construct the tree.

Finally, the Inorder subprogram traverses the structure in "inorder" once the tree has been built. "Inorder" simply means that every node is printed in between that node's subtrees. This traversal mechanism, as you will see, is quite short, and is a truly elegant expression of the task being performed.

Here are the Init_tree, Buildstring, and Inorder subprograms (Buildnum is so similar to Buildstring that it isn't necessary to list it too):

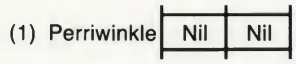
```
200  SUB Init_tree(INTEGER Root,Tree(*))
210  COM /Tree/ INTEGER Nil,Left,Right
220  Nil=0
230  Left=1
240  Right=2
250  Root=1
260  Tree(Root,Left)=Nil
270  Tree(Root,Right)=Nil
280  SUBEND
281  !
290  SUB Buildstring(INTEGER Root,Tree(*),Index,A$(*))
300  COM /Tree/ INTEGER Nil,Left,Right
310  IF A$(Index)<=A$(Root) THEN      ! Search the left subtree
320     IF Tree(Root,Left)=Nil THEN   ! Once a leaf is found (link is
330        Tree(Root,Left)=Index      !  nil) point to the new node
340        Tree(Index,Left)=Nil       !  (Index) with the leaf's left
350        Tree(Index,Right)=Nil      !  pointer and set up the new
351                                   !  node as a leaf.
360     ELSE
370        Buildstring(Tree(Root,Left),Tree(*),Index,A$(*))
380     END IF
390  ELSE                             ! Search the right subtree
400     IF Tree(Root,Right)=Nil THEN ! Once a leaf is found (link is
410        Tree(Root,Right)=Index     !  nil) point to the new node
420        Tree(Index,Left)=Nil       !  from the right pointer instead
430        Tree(Index,Right)=Nil      !  of the left.
440     ELSE
450        Buildstring(Tree(Root,Right),Tree(*),Index,A$(*))
460     END IF
470  END IF
480  SUBEND
481  !
490  SUB Inorder(INTEGER Root,Tree(*),Name$(*),INTEGER Units(*),REAL Failures(*)
)
500  COM /Tree/ INTEGER Nil,Left,Right
510  IF Root<>Nil THEN
520     Inorder(Tree(Root,Left),Tree(*),Name$(*),Units(*),Failures(*))
530     PRINT Name$(Root),Units(Root),Failures(Root)
540     Inorder(Tree(Root,Right),Tree(*),Name$(*),Units(*),Failures(*))
550  END IF
560  SUBEND
```

Let's step through a sample input stream and see how the tree is constructed using the Buildstring subprogram:

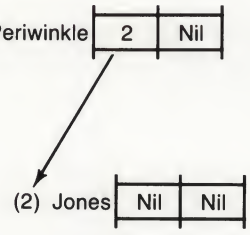
`Name$(*)`

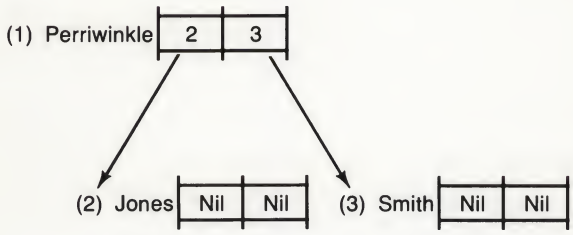
1. Perriwinkle
2. Jones
3. Smith
4. Snodgrass
5. Figby
6. Brown
7. Thompson
8. Richards
9. Hughes
10. Davenport

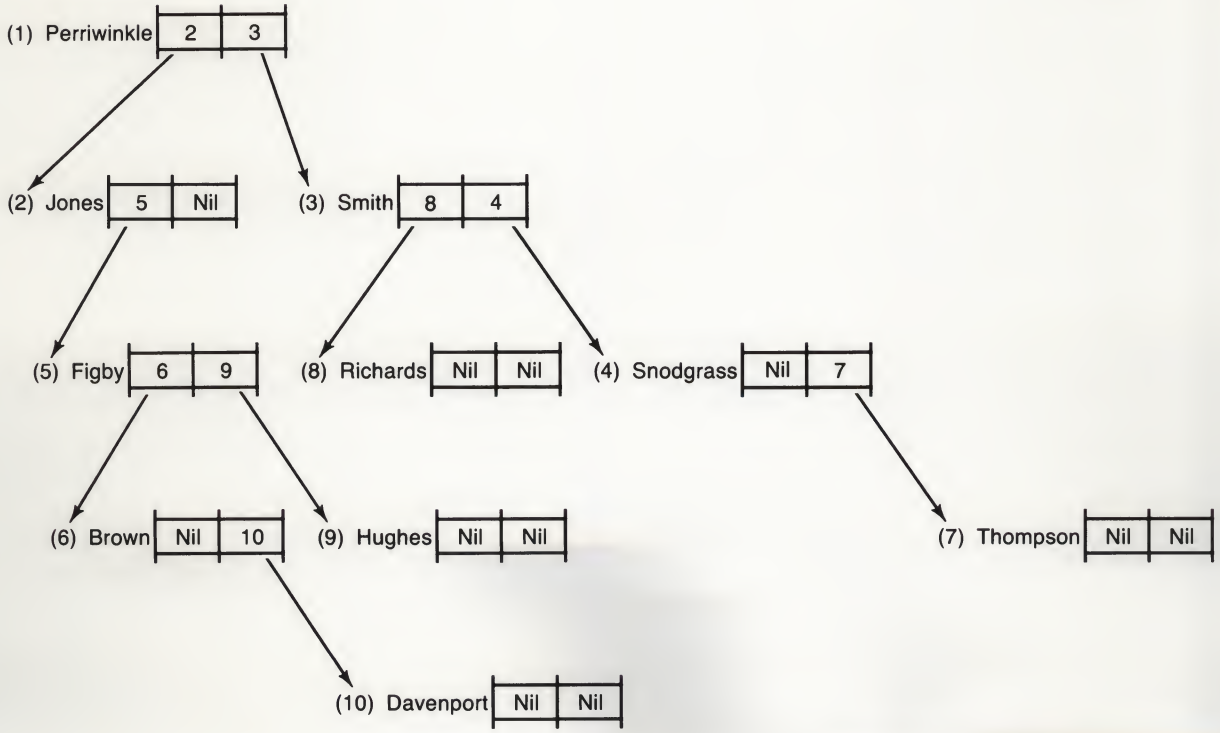
Tree structure after Init_tree is executed



Tree structure after subsequent insertions into the tree by the Buildstring subprogram:







These three subprograms illustrate several points that were discussed in this chapter:

They share a labeled COM block which is not declared in the main program, nor in the Report program. The information in the COM block was only relevant to the the three subprograms, yet the programs never called each other — they were all called from Report.

Both the Inorder and Buildstring subprograms are recursive — they call themselves. This technique was an appealing way to solve the problem because of the recursive nature of the data structure. (Many types of data structures are recursively defined.)

The use of subprograms to build and traverse the data structure turned out to execute faster than a sort subprogram which physically moved the items in the three fields into a given order based on sorting one of the arrays. (The difference was about 40% using Donald Shell's algorithm[5].)

The method of Top-Down Design led to the orderly design, creation, and testing of each subprogram, module by module, layer by layer. Communication paths and data structures/ types were forced to be clearly defined at each step of the way.

References:

1 Wirth, Niklaus, "Program Development by Stepwise Refinement", *Communications of the ACM*, April 1971, Vol. 14, No. 4, pp. 221-227

2 Yourdan, Edward, *Techniques of Program Structure and Design*, (Prentice-Hall, Englewood Cliffs, NJ, 1975)

3 Dahl, Dijkstra, & Hoare, *Structured Programming* (Academic Press, New York, 1972)

4 Knuth, Donald E., *The Art of Computer Programming*, Vol. 1, Fundamental Algorithms (Addison-Wesley, Reading, Mass, 1973), pp. 308-309,316-317

5 Knuth, Donald E., *The Art of Computer Programming*, Vol. 3, Sorting and Searching (Addison-Wesley, Reading, Mass, 1973), pp. 84-85

Chapter 7
Data Storage and Retrieval

This chapter describes some useful techniques for storing and retrieving data. First we describe how to store and retrieve data that is part of the BASIC program. With this method, DATA statements specify data to be stored in the memory area used by BASIC programs; thus, the data is always kept with the program, even when the program is stored in a mass storage file. The data items can be retrieved by using READ statements to assign the values to variables. This is a particularly effective technique for small amounts of data that you want to maintain in a program file.

For larger amounts of data, mass storage files are more appropriate. Files provide means of storing data on mass storage devices. The two types of data files available with Series 200 computers — ASCII and BDAT files — are described in this chapter. A number of different techniques for accessing data in BDAT files are described in detail. General disc structure is also described.

Series 200 computers can use a number of different mass storage devices, including disc drives (internal and external), SRM systems, and magnetic bubble memories. This chapter gives guidelines for accessing many kinds of devices.

Storing Data in Programs

This section describes a number of ways to store values in memory. In general, these techniques involve using program variables to store data. The data are kept with the program when it is stored on a mass storage device (with STORE and SAVE). These techniques allow extremely fast access of the data. They provide good use of the computer's memory for storing relatively small amounts of data.

Storing Data in Variables

Probably the simplest method of storing data is to use a simple assignment, such as the following LET statements:

```
100    LET Cm_per_inch=2.54
110    Inch_per_cm=1/Cm_per_inch
```

The data stored in each variable can then be retrieved simply by specifying the variable's name. This technique works well when there are only a relatively few items to be stored or when several data values are to be computed from the value of a few items. The program will execute faster when variables are used than when expressions containing constants are used; for instance, using the variable `Inch_per_cm` in the preceding example would be faster than using the constant expression `1/2.54`. In addition, it is easier to modify the value of an item when it appears in only one place (i.e., in the LET statement).

Data Input by the User

You also can assign values to variables at run-time with the INPUT and LINPUT statements as shown in the following examples.

```
100    INPUT "Type in the value of X, please.",Id
  .
  .
  .
200    DISP "Enter the value of X, Y, and Z."
 10    LINPUT Response$
```

Note that with this type of storage, the values assigned to the corresponding variables are *not* kept with the program when it is stored; they must be entered each time the program is run. This type of data storage can be used when the data are to be checked or modified by the user each time the program is run. As with the preceding example, the data stored in each variable can then be retrieved simply by specifying the variable's name.

Using DATA and READ Statements

The DATA and READ statements provide another technique for storing and retrieving data from the computer's read/write (R/W) memory. The DATA statement allows you to store a stream of data items in memory, and the READ statement allows you retrieve data items from the stream.

You can have any number of READ and DATA statements in a program in any order you want. When you RUN a program, the system concatenates all DATA statements in the same context into a single "data stream." Each subprogram has its own data stream. The following DATA statements distributed in a program would produce the following data stream.

```
100 DATA 1,A,50
    .
    .

200 DATA "BB",20,45
    .
    .

300 DATA X,Y,77
```

DATA STREAM:

1	A	50	BB	20	45	X	Y	77

As you can see from the example above, a data stream can contain both numeric and string data items; however, each item is stored as if it were a string.

Each data item must be separated by a comma and can be enclosed in optional quotes. Strings that contain a comma, exclamation mark, or quote mark must be enclosed in quotes. In addition, you must enter two quote marks for every one you want in the string. For example, to enter the string QUOTE"QUO"TE into a data stream, you would write:

```
100 DATA "QUOTE""QUO""TE"
```

To retrieve a data item, assign it to a variable with the READ statement. Syntactically, READ is analogous to DATA; but instead of a data list, you use a variable list. For instance, the statement:

```
100 READ X,Y,Z$
```

would read three data items from the data stream into the three variables. Note that the first two items are numeric and the third is a string variable.

Numeric data items can be READ into either numeric or string variables. If the numeric data item is of a different type than the numeric variable, the item is converted (i.e., REALs are converted to INTEGERs, and INTEGERs to REALs). If the conversion cannot be made, an error is returned. Strings that contain non-numeric characters must be READ into string variables. If the string variable has not been dimensioned to a size large enough to hold the entire data item, the data item is truncated.

The system keeps track of which data item to READ next by using a "data pointer". Every data stream has its own data pointer which points to the next data item to be assigned to the next variable in a READ statement. When you run a program segment, the data pointer is placed initially at the first item of the data stream. Every time you READ an item from the stream, the pointer is moved to the next data item. If a subprogram is called by a context, the position of the data pointer is recorded and then restored when you return to the calling context.

Starting from the position of the data pointer, data items are assigned to variables one by one until all variables in a READ statement have been given values. If there are more variables than data items, the system returns an error, and the data pointer is moved back to the position it occupied before the READ statement was executed.

Examples

The following example shows how data is stored in a data stream and then retrieved. Note that DATA statements can come after READ statements even though they contain the data being READ. This is because DATA statements are linked during program pre-run, whereas READ statements aren't executed until the program actually runs.

```
10  DATA November,26
20  READ Month$,Day,Year$
30  DATA 1981,"The date is"
40  READ Str$
50  Print Str$;Month$,Day,Year$
60  END


The date is November 26 1981
```

Storage and Retrieval of Arrays

In addition to using READ to assign values to string and numeric variables, you can also READ data into arrays. The system will match data items with variables one at a time until it has filled a row. The next data item then becomes the first element in the next row. You must have enough data items to fill the array or you will get an error. In the example below, we show how DATA values can be assigned to elements of a 3-by-3 numeric array.

```
10    OPTION BASE 1
20    DIM Example(3,3)
30    DATA 1,2,3,4,5,6,7,8,9,10,11
40    READ Example(*)
50    PRINT USING "3(K,X),/";Example(*)
60    END


 1 2 3
 4 5 6
 7 8 9
```

The data pointer is left at item 10; thus, items 10 and 11 are saved for the next READ statement.

Moving the Data Pointer

In some programs, you will want to assign the same data items to different variables. To do this, you have to move the data pointer so that it is pointing at the desired data item. You can accomplish this with the RESTORE statement. If you don't specify a line number or label, RESTORE returns the data pointer to the first data item in the data stream. If you do include a line identifier in the RESTORE statement, the data pointer is moved to the first data item in the first DATA statement at or after the identified line. The example below illustrates how to use the RESTORE statement.

```
100   DIM Array1(1:3)    ! Dimensions a 3-element array.
110   DIM Array2(0:4)    ! Dimensions a 5-element array.
120   DATA 1,2,3,4       ! Places 4 items in stream.
130   DATA 5,6,7         ! Places 3 items in stream.
140   READ A,B,C         ! Reads first 3 items in stream.
150   READ Array2(*)     ! Reads next 5 items in stream.
160   DATA 8,9           ! Places 2 items in stream.
170                      !
180   RESTORE            ! Re-positions pointer to 1st item.
190   READ Array1(*)     ! Reads first 3 items in stream.
200   RESTORE 140        ! Moves data pointer to item "8".
210   READ D             ! Reads "8".
220                      !
230   PRINT "Array1 contains:";Array1(*);" "
240   PRINT "Array2 contains:";Array2(*);" "
250   PRINT "A,B,C,D equal:";A;B;C;D
260   END
```

```
Array1 contains: 1 2 3
Array2 contains: 4 5 6 7 8
A,B,C,D equal: 1 2 3 8
```

Mass Storage Tutorial

The rest of this chapter describes the second general class of data storage and retrieval methods — that of using mass storage devices. This material is broken up into two main parts. The first part presents a general tutorial regarding disc and file structures. The second part presents techniques for accessing mass storage devices and files with BASIC statements. If you are anxious to "get going," turn to "Mass Storage Techniques" (about 10 pages ahead) and refer back to the tutorial as needed.

What Is Mass Storage?

As the adjective "mass" suggests, mass storage devices are data-storage devices which are generally capable of storing "large" amounts of data. Just how much data constitutes a large amount depends on the device itself. Most mass storage devices are capable of storing on the order of hundreds of thousands to several million items.

Besides having the ability to store data, mass storage devices are capable of providing means for keeping data organized so that logical groups may be accessed systematically and efficiently. Data items are organized into logical groups of data known as *files*; a file is merely a collection of data items. Mass storage *volumes* are composed of one or more files. On most HP mass storage devices, a volume consists of all files on the mass storage *media*; mass storage media are the actual physical means by which data are stored. For instance, the media used by the internal drives of the Model 226 and Model 236 consist of magnetic particles on a plastic disc which can be magnetized to store data.

The Structure of Model 226/236 Discs

This section describes the specific structure of discs used with built-in Model 226/236 drives. This specific structure is similar to the general structure of all HP discs. Some of the information in this section (directory format and disc interleave) is useful to the advanced programmer; however, it is not a prerequisite for general mass storage usage.

Most HP mass storage devices use magnetic discs as their storage media. Magnetic discs possess features of both records and recording tape. Like a record, it is circular, has tracks (most have tracks on both sides), and rotates. Like a tape, data is written and read by an electromagnetic head.

The internal disc drive(s) of the Model 226 and 236 use 5.25-inch diameter, flexible disc media. These flexible discs have two usable sides with 33 *tracks* on each side. Each track is further divided into sixteen *sectors*, each of which contain 256 contiguous bytes of data. A track, therefore, contains 4096 bytes, and an entire disc is capable of holding over 270,000 bytes of information. However, some of this space is reserved for system use and cannot be used for data storage.

In addition to the internal disc drive(s), you can also use external drives with your computer. There are several types of drives that are compatible with HP Series 200 computers. Some mass storage devices use the same 5.25-inch diameter flexible discs as the internal drives, while others use 3.5-inch or 8-inch flexible discs and/or hard discs. We describe how to access external drives later in this chapter.

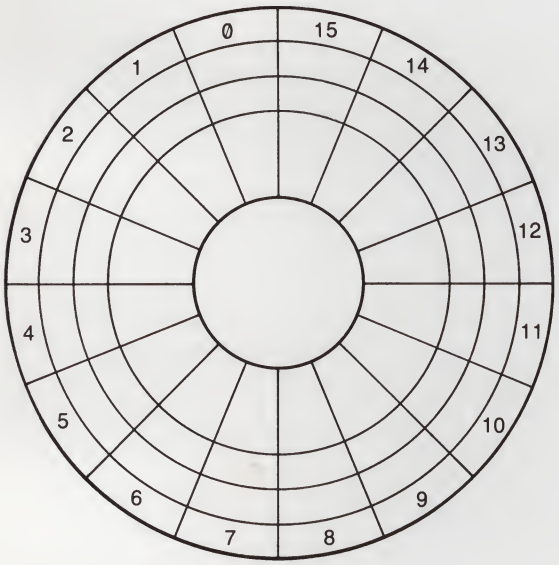
As previously mentioned, information is stored on a disc by placing it in a file. A file is a logical unit that occupies a certain number of sectors on the storage media. Files can be used to store either programs or data. In this chapter, we are concerned primarily with data files, although parts of the discussion relate to program files as well.

When you SAVE a program, the system automatically creates an ASCII file of sufficient size to store the program in its "source form" — as ASCII characters. When you create a data file with the CREATE ASCII or CREATE BDAT statements, you give it a name and specify how much disc space should be reserved for it. In general, the only limit to the size of a file is the amount of contiguous, unreserved space left on the media, since files cannot span disc volumes.

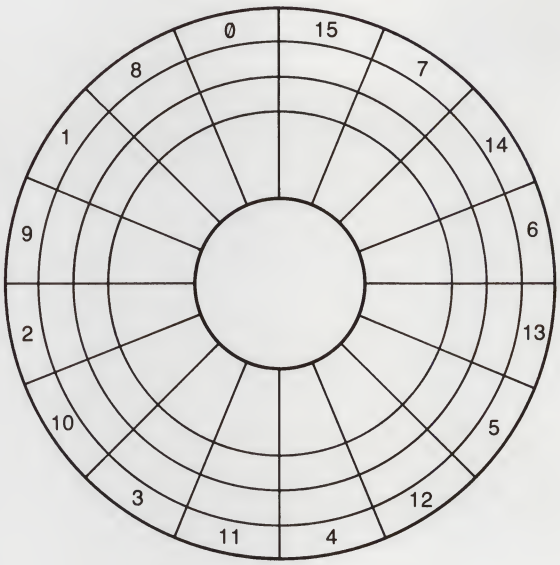
Let's take a brief look at how the the system accesses the information in the file. (More complete details of file access will be described later.) In order to access the information in a file, you assign an I/O path name to the file's name. When this I/O path is used to send and receive information to and from the file, the system handles the details of the access operation. The important point here is that the system *always* reads and/or writes an entire sector of data whenever a disc is accessed. For instance, if only one byte of data is to be written in a file, an entire sector must be read, the single byte changed, and the (modified) sector re-written.

Disc Interleave

The INITIALIZE statement allows you to specify an interleave factor. Interleaving a disc causes the sectors on each track to be numbered according to a specified interval. An interleave factor of 1 causes sectors to be numbered consecutively. A factor of 2, on the other hand, tells the system to skip every other sector. The following drawing shows how these interleaves are implemented on the 5.25-inch flexible discs used by the Model 226 and Model 236 internal drives.



A track of a disc with interleave factor of 1.



A track of a disc with interleave factor of 2.

The system numbers each sector on each track according to the pattern specified by the interleave factor. All tracks on a disc have the same interleave factor.

The purpose of disc interleave is to increase data-transfer rates, as demonstrated in the following example. Suppose that we are entering data from a disc; suppose also that the data have formats which are different from the computer's internal data formats. Consequently, after each item is read, the computer must change the data from the disc's data format to the computer's internal data format. Note that during this processing time the disc is still spinning.

If the processing of all items in a sector takes more time than the disc drive takes to reach the next sector, the computer must wait (slightly longer than one full revolution of the disc) until the next sector again comes under the head. By interleaving a disc, you can allow for this processing time, which results in faster transfer rates.

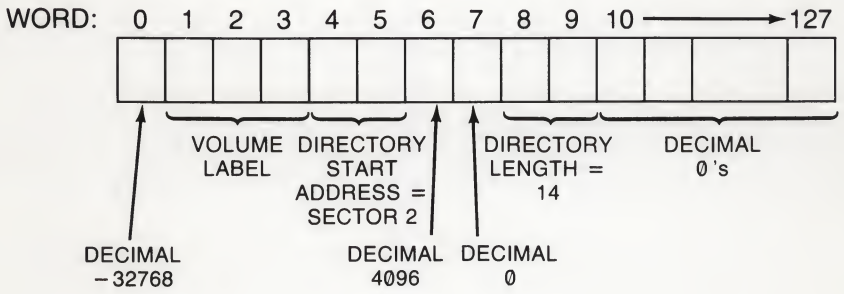
The default interleave factor for each type of drive is designed to give maximum transfer rates for LOAD and STORE operations (and for OUTPUT and ENTER of numeric arrays using BDAT files with the FORMAT OFF attribute). The default interleave factor is 1 for internal drives of the Model 226 and Model 236. For other HP drives, the optimum interleave may differ. All default interleave factors are shown later in this chapter.

If you use the default interleave factors, you should be aware that discs initialized on one drive may show a performance degradation if used on another drive. For instance, suppose you INITIALIZE a disc on the internal drive (default interleave factor is 1) and then use that disc in an HP 82901 drive (default interleave factor of 4). You probably will get slower transfer rates than if the default interleave for the HP 82901 had been used. *In summary, if you need maximum transfer rates, experiment to determine the optimal interleave for your particular application.*

When estimating optimal interleave factors, it is better to use a factor too large than one too small. For instance, suppose that an interleave of 2 is optimal for a particular operation. If an interleave of 1 is improperly chosen, the operation is slowed down approximately by a factor of eight. On the other hand, using an interleave of 3 only slows the operation approximately by a factor of two. If in doubt about which of two interleave factors is optimal for a particular situation, it is *generally* better to choose the *larger.*

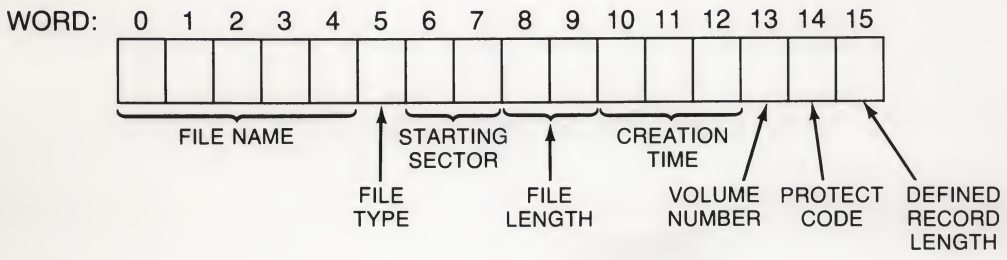
Volume Label

The first sector on every disc contains information about the disc volume. It contains the name of the volume, the starting address of the directory, and the length of the directory. The figure below shows the values and locations of this information for LIF-compatible discs.



Directory

A directory is an index of all files on the disc. Every disc volume has a directory. On Model 226/236 internal disc volumes initialized with BASIC, the directory occupies sectors 2 thru 15. Other discs or discs initialized in other languages may have a different directory size. However, the general structure of the directory entries is the same. In this directory, there is a 16-word entry for every file; each directory sector can thus hold 8 entries. Since this directory contains 14 sectors, it can hold up to 112 entries; therefore, the limit on the number of files in a volume is 112. The figure below shows the format of a directory entry.



File Name — Every file is given a name when it is created. File names can be up to 10 characters long.

File Type — BASIC supports five file types: ASCII, BDAT, BIN, PROG and SYSTM. We discuss BDAT and ASCII data type files later in this chapter.

Starting Sector — The address of the file's first sector. Files always start at the beginning of a sector.

Length of File — Length is given in sectors. If a file ends in the middle of a sector, the whole sector is counted.

Time of Creation — Not used. Entry is filled with zeros.

Volume Number — The least significant 15 bits give the volume number, which is always 1 on Series 200 computers. The most significant bit tells whether the file is continued in another volume (0 for yes, 1 for no). Since files on these computers cannot span volumes, this bit is always set to 1.

Protect Code — Any BDAT, BIN or PROG type file can be given a two-character protect code with the PROTECT statement. ASCII and SYSTM type files cannot be protected. This word is always 0 with ASCII files; it has a different use with SYSTM files. Protect codes are discussed later in this chapter.

Defined-Record Length — If a file is of BDAT type, it can have defined records from 1 thru 65 534 bytes long. This is described in detail later in this chapter. Defined records in all other file types are always one sector (256 bytes) long. The defined record length is encoded into word 15 as length/2. (If length = 1, then word 15 = 0.) This word is always 0 with ASCII files; it has a different use with SYSTM files.

The Structure of Data Files

There are two file types that you can use to store data: BDAT and ASCII. BDAT files have several advantages: they allow more flexibility in data formats and access methods, allow faster transfer rates, and are generally more space-efficient than ASCII data files. They can be randomly or serially accessed, and they allow data to be stored in either ASCII format, internal format, or a specialized format (defined by the user with IMAGE statements).

ASCII files allow *only* serial access and *only* ASCII format. They have these advantages: the files are compatible with other HP computers that support this file type, the format provides very compact storage for string data, and there is no chance of reading the contents into the wrong data type (a problem for BDAT files). The full name of ASCII files is "LIF ASCII". LIF stands for Logical Interchange Format, a directory and data storage format that is used by many HP computer divisions. Understanding the characteristics of each file type will help you choose the best one for your specific application.

BDAT Files

BDAT files are designed to be storage-space efficient, have high data-transfer rates, and allow *both* random and serial access. Random access means that you can directly read from and write to any record within the file, while serial access only permits you to access the file from the beginning. Serial access can waste a lot of time if you're trying to access data at the end of a file. On the other hand, if you want to access the entire file sequentially, you are better off using serial access than random access. BDAT files can be accessed both randomly and serially, while ASCII files can only be accessed serially.

BDAT files allow you to store and retrieve data using internal format, ASCII format, or user-defined formats. With internal format, items are represented with the same format the system uses to store data in internal computer memory[1]. With ASCII format, items are represented by ASCII characters. User-defined formats are implemented with programs that employ OUTPUT and ENTER statements that reference IMAGE specifiers. Complete descriptions of ASCII and user-defined formats are given in Chapters 4, 5, and 10 of *BASIC Interfacing Techniques.*

In most applications, you will use internal format for BDAT files. Unless we specify otherwise, you can assume that when we talk about retrieving and storing data in BDAT files, we are also talking about internal format. This format is synonymous with the FORMAT OFF attribute, which is described later in this chapter.

Because BDAT files use almost the same format as internal memory, very little interpretation is needed to transfer data from the computer to a BDAT file, or vice versa. BDAT files, therefore, not only save space but also time.

Data stored in internal format in BDAT files require the following number of bytes per item:

INTEGER	2 bytes
REAL	8 bytes
String	1 byte per character (plus 1 pad byte if the string length is an odd number), plus a 4-byte length header

1 Actually, the format for BDAT files is slightly different than internal format. Instead of using a 2-byte length header for strings, BDAT files use a 4-byte length header. Besides this, the two formats are identical, so we refer to both as "internal".

INTEGER numbers are represented in BDAT files by using a 16-bit, two's-complement notation, which provides a range −32 768 thru 32 767. If bit 15 (the MSB) is 0, the number is positive. If bit 15 equals 1, the number is negative; the value of the negative number is obtained by changing all ones to zeros, and all zeros to ones, and then adding one to the resulting value.

Examples

Binary Representation	Decimal Equivalent
00000000 00010111	23
11111111 11101000	−24
10000000 00000000	−32768
01111111 11111111	32767
11111111 11111111	−1
00000000 00000001	1
00100011 01000111	9031
11011100 10111001	−9031

REAL numbers are stored in BDAT files by using their internal format: the IEEE-standard, 64-bit, floating-point notation. Each REAL number is comprised of two parts: an exponent (11 bits), and a mantissa (53 bits). The mantissa uses a sign-and-magnitude notation. The sign bit for the mantissa is not contiguous with the rest of the mantissa bits; it is the most significant bit (MSB) of the entire eight bytes. The 11-bit exponent is offset by 1 023 and occupies the 2nd through the 12th MSB's. Every REAL number is internally represented by the following equation. (Note that the mantissa is in binary notation):

$$-1^{\text{mantissa sign}} \times 2^{\text{exponent} - 1023} \times 1._{\text{mantissa}}$$

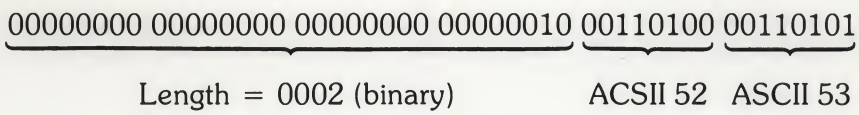
The figure below shows how the real number "1/3" would be stored in a BDAT file.

Byte	1	2	3	4	...	8
Decimal value of character	63	213	85	85	...	85
Binary value of characters	00111111	11010101	01010101	01010101	...	01010101

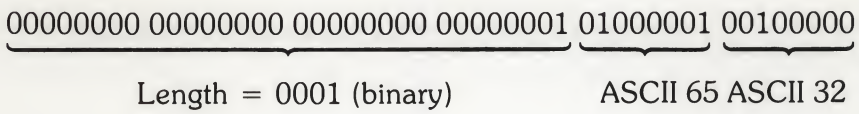
mantissa sign — exponent — mantissa

String data are stored in BDAT files in their internal format (plus two additional, leading bytes of length header, which are always 0 for Series 200 computers). Every character in a string is represented by one byte which contains the character's ASCII code. The four-byte length header contains a value that specifies the length of the string. If the length of the string is odd, a pad character is appended to the string to get an even number of characters; however, the length header does not include this pad character.

Examples
If stored as a string value, the number "45" would be:



The string "A" would be stored:

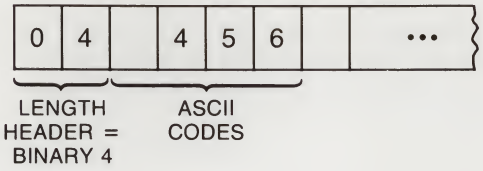


In this case, the space character (ASCII code 32) is used as the pad character; however, not all operations use the space as the pad character.

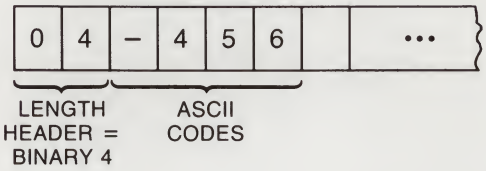
When using the ASCII data format for BDAT files, all data items are represented with ASCII characters. With user-defined formats, the image specifiers referenced by the OUTPUT or ENTER statement are used to determine the data representation. Using both of these formats with BDAT files produce results identical to using them with devices. The entire subject is described fully in Chapters 4, 5, and 10 of *BASIC Interfacing Techniques*. The topic of advanced transfer techniques for BDAT files is described in Chapter 11 of the same manual. Refer to this material after reading "Mass Storage Techniques" later in this chapter.

ASCII Files

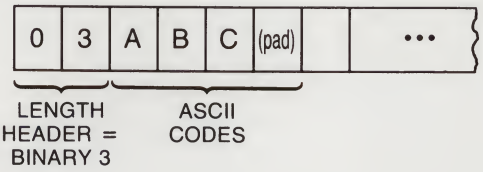
You have already been introduced to ASCII files as a way to SAVE programs. ASCII files can also be used to store data. In an ASCII file, every data item, whether string or numeric, is represented by ASCII characters; one byte represents one ASCII character. Each data item is preceded by a two-byte length header which indicates how many ASCII characters are in the item. However, there is no "type" field for each item; data items contain no indication (in the file) as to whether the item was stored as string or numeric data. For instance, the number 456 would be stored as follows in an ASCII file:



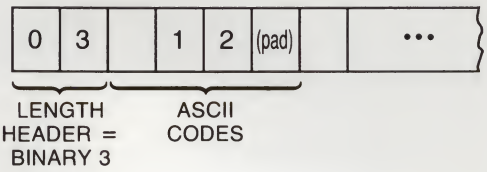
Note that there is a space at the beginning of the data item. This signifies that the number is positive. If a number is negative, a minus sign precedes the number. For instance, the number −456, would be stored as follows:



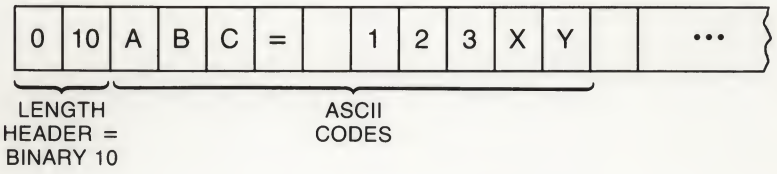
If the length of the data item is an odd number, the system "pads" the item with a space to make it come out even. The string "ABC", for example, would be stored as follows:



There is often a relatively large amount of overhead for numeric data items. For instance, to store the integer 12 in an ASCII file requires the following six bytes:



Similarly, reading numeric data from an ASCII file can be a complex and relatively slow operation. The numeric characters in an item must be entered and evaluated individually by the system's "number builder" routine, which derives the number's internal representation. (Keep in mind that this routine is called automatically when data are entered into a numeric variable.) For example, suppose that the following item is stored in an ASCII file:



Although it may seem obvious that this is not a numeric data item, the system has no way of knowing this since there is no type-field stored with the item. Therefore, if you attempt to enter this item into a numeric variable, the system uses the number-builder routine to strip away all non-numeric characters and spaces and assign the value 123 to the numeric variable. When you add to this the intricacies of real numbers and exponential notation, the situation becomes more complex. For more information about how the number builder works, see Chapter 5 of *BASIC Interfacing Techniques.*

Because ASCII files require so much overhead (for storage of "small" items), and because retrieving numeric data from ASCII files is sometimes a complex process, they are not the preferred file type. However, as we mentioned before, ASCII files are interchangeable with many other HP products.

In this chapter, we refer to the data representation described above as ASCII-file format. As mentioned earlier, you can also store data in BDAT files in ASCII format (by using the FORMAT ON attribute). However, be careful not to confuse ASCII-file format with the ASCII data format.

In general, you should only use ASCII files when you want to transport data between HP Series 200 computers and other HP machine(s). There may be other instances where you will want to use ASCII files, but you should be aware that they cause a noticable performance degradation compared to BDAT files.

Mass Storage Techniques

This section presents BASIC programming techniques useful for accessing mass storage devices and files. The first section gives a brief introduction to the steps you might take to store data in a file. Subsequent sections describe further details of these steps. If you feel that you need additional background information while reading this material, refer to the preceding tutorial section.

Initializing a Disc

Before a disc is used for the first time, it must be initialized. If the disc has already been initialized on a LIF-compatible device, and it contains data you wish to retain, then it can be used on a Series 200 computer without initialization. However, if a previously initialized disc does not have any data on it (or you don't need the data on it), it might be advantageous to re-initialize it on your computer to get maximum performance. The point is this: a disc must be properly initialized before your computer can use it, but initializing a disc **destroys all the data on the disc**.

The following steps show a typical initialization process using an internal disc drive of a Model 226 or 236 as an example. The procedure for initializing external discs is very similar, but specific details will change. For example, an external disc drive will have a different specifier (not :INTERNAL), may have a different write-protect convention, and will probably take a different length of time to initialize. For other examples, look in your operating manual or check the *BASIC Language Reference* under "MASS STORAGE IS" and "INITIALIZE"

To initialize a 5.25-inch disc on an internal disc drive, follow these steps:

1. Make sure that the disc does not contain any important data or programs. Many types of computers and word processors use similar discs. When a disc is initialized, all the data on it is destroyed!
2. Ensure that the disc is not "write protected". The disc envelope has a small notch on one side. When this notch is open, the computer is allowed to write on the disc. If this notch is covered, data may be read from the disc, but recording is not allowed. Trying to initialize a write-protected disc results in error number 83.
3. Be sure the disc is properly inserted in the right-hand disc drive.
4. Execute `INITIALIZE ":INTERNAL"`. This command tells the computer to erase all data from the disc, format it for use in your computer, check the quality of the media, and create the directory area.

An initialize operation takes about three minutes. The CRT displays the system's progress during this operation. When the initialization is complete, the run light turns off and a message similar to the following is displayed.

```
INITIALIZE: TRACK 32, SIDE 1, SPARED 0
```

During initialization, the disc is checked for bad areas on the tracks. If a bad spot is found, that particular track is rejected (or "spared"). If track 0 is bad or more than four tracks are spared, the initialize operation fails and error number 66 is issued. Although a disc with less than 5 rejected tracks can be used, rejected tracks are an indication that the magnetic surface is not in very good condition.

After the initialization has completed successfully, the disc is ready for storing programs and data.

Disc Labels

After you initialize the disc, you may want to give it a label. The PRINT LABEL statement prints the label in the disc directory. Once the label is there, a READ LABEL statement can retrieve it. The disc label is included in a CATalog of the disc.

For example, to give the disc in the INTERNAL disc drive the label VOL1, execute the following:

```
PRINT LABEL "VOL1" TO ":INTERNAL"
```

To read the label, enter:

```
10 READ LABEL Name$ FROM ":INTERNAL"
```

Disc labels are useful in many cases. When a program asks the operator to insert a particular disc in the disc drive, the program can read the label to insure the correct disc was inserted.

```
1000   Insert:  !Insert disc
1010   DISP "Insert disc VOL1 in the INTERNAL disc drive then press CONT"
1020   PAUSE
1030   READ LABEL Label$ FROM ":INTERNAL"
1040   IF Label$<>"VOL1" THEN
1050      DISP "You have inserted an incorrect disc"
1060      BEEP
1070      WAIT 3
1080      GOTO Insert
1090   END IF
1100   RETURN
```

If several disc drives are connected to the computer, a program can read the label of each disc to find the disc it needs to access.

```
2000   !Read the disc labels
2010   READ LABEL Drive0$ FROM ":INTERNAL"
2020   READ LABEL Drive1$ FROM ":INTERNAL,4,1"
2040   SELECT 1
2050   CASE Drive0$="VOL1"
2060        ! Access files from ":INTERNAL"
2070   CASE Drive1$="VOL1"
2080        ! Access files from ":INTERNAL,4,1"
2090   CASE ELSE
2100        ! Disc not in drive
2110   END SELECT
```

Overview of Mass Storage Access

Storing data in files requires a few simple steps. The following program segment shows a simplistic example of placing several items in a data file. Assume that this program is run on a Model 236.

```
   .
   .
390  ! Specify left drive as "system" mass storage.
400  MASS STORAGE IS ":INTERNAL,4,1"
410  !
420  ! Create BDAT data file with ten (256-byte) records
430  ! on the system mass storage (left drive).
440  CREATE BDAT "File_1",10
450  !
460  ! Assign (open) an I/O path to the file.
470  ASSIGN @Path_1 TO "File_1"
480  !
490  ! Send an array of numeric values.
500  OUTPUT @Path_1;Array1(*)
510  !
520  ! Close the I/O path (may be optional).
530  ASSIGN @Path_1 TO *
   .
   .
790  ! Open another I/O path to the file.
800  ASSIGN @F_1 TO "File_1:INTERNAL,4,1"
810  !
820  ! Read data into another array (same size and type).
830  ENTER @F_1;Array2(*)
840  !
850  ! Close I/O path.
860  ASSIGN @F_1 TO *
```

Line 400 specifies the "system mass storage device," or the "default" device which is to to be used whenever a mass storage device is not explicitly specified during subsequent mass storage operations. The term *mass storage unit specifier (msus)* describes the string expression used to uniquely identify which device is to be the mass storage. In this case, ":INTERNAL,4,1" is the msus.

In order to store data in mass storage, a data file must be created (or already exist) on the mass storage media. In this case, line 440 creates a BDAT file for data storage; the file created contains 10 defined records of 256 bytes each. (Defined records and record size are discussed later in this chapter.)

The term "file specifier" describes the string expression used to uniquely identify the file. In this example, the file specifier is simply "File_1," which is the file's name. If the file is to be created (or already exists) on a mass storage device *other than the system mass storage*, the appropriate msus must be appended to the file name.

Then, in order to store data in (or retrieve data from) the file, you must assign an I/O path name to the file. Line 470 shows an example of assigning an I/O path name to the file (also called opening an I/O path to the file). Line 500 shows an array of numeric data being sent to the file through the I/O path.

The I/O path is closed after all data have been sent to the file. In this instance, closing the I/O path may have been optional, because another I/O path name is assigned to the file later in the program. (All I/O path names are automatically closed by the system at the end of the program.) Closing an I/O path to a file updates the file pointers.

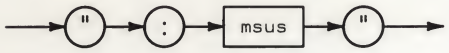
If this array of data is to be retrieved from the file, another ASSIGN statement is executed (line 800). Notice that a different I/O path name has been used; this is an arbitrary choice of names. Opening this I/O path name to the file sets the file pointer to the beginning of the file. (Re-opening the I/O path name @File_1 would have also reset the file pointer.)

Notice also that the msus is included with the file name. This shows that the mass storage device, here the left drive of the Model 236, does not have to be the current system mass storage in order to be accessed. The subsequent ENTER statement reads the data into another numeric array (which must be of the *same* data type when a BDAT file is used in this manner).

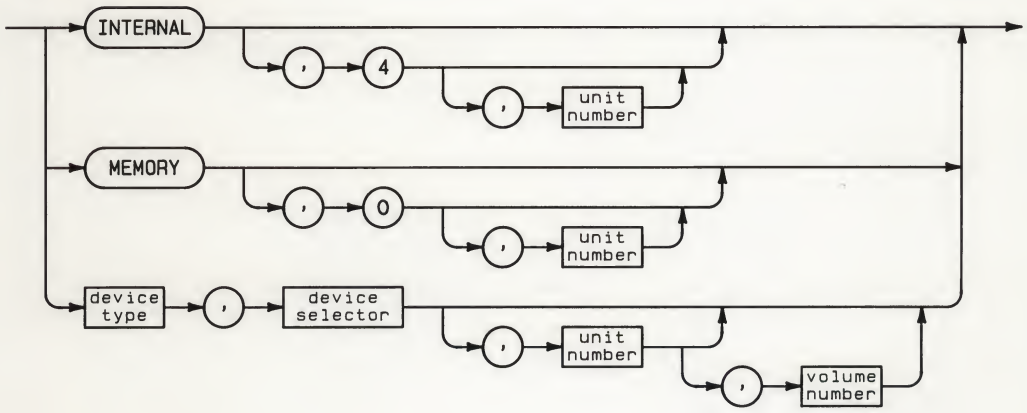
As you can see, this is a very simplistic example, in which several assumptions have been made. However, it shows the general steps you must take to access files. The rest of this section expands on these basic steps.

Media Specifiers

Once the mass storage is connected, you need a way of specifying which mass storage device to be accessed. This is done with a media specifier. The syntax for a media specifier is illustrated below.



msus



Device type — effectively describes the mass storage device to the system. The system then knows the capacity of the device, the directory structure, and other information required to determine the access method for the device. Examples are `INTERNAL`, `CS80`, and `HP82901`.

The following table shows the **valid** device types. Most device types require an option BIN for the statement to execute.

BIN Required	Device Type
none	INTERNAL MEMORY
DISC & HPIB or FHPIB	HP 9895 HP 9121 HP 9133 HP 9134 HP 9135 (5¼ inch uses HPIB not FHPIB) HP 913X
DISC & HPIB	HP 82901 HP 82902 HP 8290X
CS80 & HPIB or FHPIB	CS80 (7908, 7911, 7912, 7914, 9122...)
HP9885 & GPIO	HP 9885
SRM & DCOMM	REMOTE
BUBBLE	BUBBLE
EPROM	EPROM

Note the 98625 Card (which requires the FHPIB BIN) cannot be used with external 5¼ inch discs.

If the device type specified is not valid, the system tests the device to determine its type. There are two exceptions to this.

1. If the device selector is 0 and the device type is invalid, the device type is assumed to be MEMORY.
2. If the device type is valid and the driver BIN for the device is not loaded, the system considers the device an invalid device type.

Device selector — tells the system the select code of the interface connected to the device; if the interface is an HP-IB, it also tells the system the device's primary address. The system then knows which interface connects the device to the computer (and the device's address, if an HP-IB is used).

A device selector can be just an interface select code or a combination of select code and primary address. To derive a device selector with a primary address, multiply the interface select code by 100 and then add the address. For instance, the device selector 703 would select the device with primary address 3 which is connected to the interface at select code 7. Note that interface select code 7 is the built-in HP-IB interface; this is the interface you will probably use to attach external disc drives.

The device selector for the internal drive(s) is 4. Note that there is no default device selector for external drives; in other words, you could not use just the device type to specify an external drive.

Unit number — tells the system additional information about the device's unit-number setting. Many devices have hard-wired unit numbers, while others use the unit number to identify different portions of one disc. For instance, the unit number of the right drive of a Model 236 is 0, while it is 1 for the left drive.

Volume number — a volume is a subdivision of a unit if the device supports volumes. If the volume number is not given, the default 0 is used.

Examples
The following statements set the system mass storage to an HP 82901 drive at interface select code 7; the HP 82901 is set to primary address 0 and has a unit number of 1.

```
MASS STORAGE IS ":HP82901,700,1"
or
MASS STORAGE IS ":HP,700,1"
```

Executing the following statements catalog the disc in the HP 9121 drive at interface select code 8 with primary address 2 and unit number 0.

```
CAT ":HP9121,702"
CAT ":HP,702"
```

Again, note that there is no default device selector for external drives; in other words, you cannot use just the device type to specify an external drive.

The following statement creates an ASCII file named "Fred" on the disc in unit 3 of an HP 9134 drive, connected through interface select code 7; the device has a primary address of 0.

```
CREATE ASCII "Fred:HP9134,700,3"
```

Internal Disc Drives

There are two ways to specify which mass storage device is to be used for an operation. The first way is to use a media specifier to specify the system mass storage with the MASS STORAGE IS statement. The following statement specifies that the right drive of a Model 236 (the only drive of a Model 226) is to be system mass storage.

```
MASS STORAGE IS ":INTERNAL"
```

Now whenever you perform a mass storage operation and don't specify a mass storage device, the system will automatically use this mass storage device. This default mass storage remains effective until you either execute a new MASS STORAGE IS statement or a SCRATCH A. After a power-up or SCRATCH A, the MASS STORAGE IS device is set to the device from which the system was booted.

The second method is to specify a media specifier for the operation. For example, the following statement directs the system to catalog the disc media in the left drive (of a Model 236).

```
CAT ":INTERNAL,4,1"
```

Note that, with this second method, using a media specifier overrides the default mass storage for the particular operation but does not modify the system mass storage (i.e., the default msus).

The following statement sets the default mass storage to the left drive of a Model 236. Note that "MASS STORAGE IS" can be abbreviated as "MSI."

```
MSI ":INTERNAL,4,1"
```

Executing this statement on a computer other than a Model 236 results in an error, since it has no left drive.

The "4" parameter is the internal drive's interface select code, and the "1" is the unit number of the left drive. It is possible (but redundant) to use the parameters which specify the right-hand drive, since the default values also specify the right drive. For example, all of the following statements direct the system to catalog the disc in the right drive.

```
CAT ":INTERNAL,4,0"
CAT ":INTERNAL,4"
CAT ":INTERNAL"
CAT ":,4"
```

You can also specify external mass storage devices by using the proper media specifier. The next section describes the following topics: external HP drives which are supported by the BASIC language of Series 200 Computers and non-disc mass storage devices such as magnetic bubble memories. If you aren't going to use external mass storage, you can skip to the section that describes file access.

External Disc Drives

HP Series 200 computers support a number of external HP disc drives. The following table shows the external drives which can be used. The Total Storage Capacity values given are on a per-media basis *before* the system writes directory information on the media.

HP Device	Type of Media	Total Storage Capacity: Bytes	Files
HP 7908	hard disc (optional tape backup)	16 576 000	2 584
HP 7911	hard disc (optional tape backup)	28 114 944	4 392
HP 7912	hard disc (optional tape backup)	65 601 536	10 248
HP 7914	hard disc (optional tape backup)	132 120 576	20 640
HP 9121S	3.5-inch flexible disc (single-drive)	270 336	112
HP 9121D	3.5-inch flexible disc (dual-drive)	270 336	112
HP 9133 (flexible disc for all versions)	3.5-inch flexible disc	270 336	112
HP 9133A	5.25-inch hard disc	1 152 000 per unit	464 per unit
HP 9133A, Option 010	5.25-inch hard disc	4 825 088	752
HP 9133B	5.25-inch hard disc	9 581 920	1 512
HP 9133V	5.25-inch hard disc	4 825 088	752
HP 9133V, Option 004	5.25-inch hard disc	1 152 000 per unit	464 per unit
HP 9133XV	5.25-inch hard disc	14 522 880	2 264
HP 9133XV, Option 010	5.25-inch hard disc	9 681 920	1 512
HP 9134A	5.25-inch hard disc	1 152 000 per unit	464 per unit
HP 9134A, Option 010	5.25-inch hard disc	4 825 088	752
HP 9134B	5.25-inch hard disc	9 681 920	1 512
HP 9134XV	5.25-inch hard disc	14 522 880	2 264
HP 9134XV, Option 010	5.25-inch hard disc	9 681 920	1 512
HP 9135 (flexible disc for all versions)	5.25-inch flexible disc	270 336	112
HP 9135A	5.25-inch hard disc	1 152 000	464
HP 9135A, Option A	5.25-inch hard disc	4 825 088	752
HP 9885	8-inch flexible disc	483 840	224
HP 9895	8-inch flexible disc (single-sided media)	483 840	224
HP 9895	8-inch flexible disc (double-sided media)	1 152 000	464
HP 82901	5.25-inch flexible disc (dual-drive)	270 336	112
HP 82902	5.25-inch flexible disc (single-drive)	270 336	112

You can have nearly any number and any combination of these drives connected to your computer; the only restriction is imposed by the number of and the limitations of the interface(s) used to connect the drives to the computer.

The next sections summarize configuring and accessing external devices currently available. The following table summarizes pertinent information regarding the available drives.

All of the entries in the table may not make complete sense to you at this time; however, once you connect your drive, you will be able to better understand the data and use the information as a handy quick reference.

HP Device	Device Type	Interface Required	Factory Settings: Address	Factory Settings: Unit No.	Default Interleave
HP 7908 (h)	CS80	HP-IB or fast HP-IB	0	0	1
HP 7908 (t)	CS80	HP-IB or fast HP-IB	0	1	—
HP 7911 (h)	CS80	HP-IB or fast HP-IB	0	0	1
HP 7911 (t)	CS80	HP-IB or fast HP-IB	0	1	—
HP 7912 (h)	CS80	HP-IB or fast HP-IB	0	0	1
HP 7912 (t)	CS80	HP-IB or fast HP-IB	0	1	—
HP 7914 (h)	CS80	HP-IB or fast HP-IB	0	0	1
HP 9121S (t)	HP9121	HP-IB or fast HP-IB	0	0	2
HP 9121D (l)	HP9121	HP-IB or fast HP-IB	0	0	2
HP 9121D (r)	HP9121	HP-IB or fast HP-IB	0	1	4
HP 9133 (f, all versions)	HP913X	HP-IB or fast HP-IB	2	0	2
HP 9133A (h)	HP913X	HP-IB or fast HP-IB	0	0 thru 3	9
HP 9133A, Opt. 010 (h)	HP913X	HP-IB or fast HP-IB	0	0	9
HP 9133B (h)	HP913X	HP-IB or fast HP-IB	0	0	9
HP 9133V (h)	HP913X	HP-IB or fast HP-IB	0	0	9
HP 9133V, Opt. 004 (h)	HP913X	HP-IB or fast HP-IB	0	0 thru 3	9
HP 9133 XV (h)	HP913X	HP-IB or fast HP-IB	0	0	9
HP 9133XV, Opt. 010 (h)	HP913X	HP-IB or fast HP-IB	0	0	0
HP 9134A	HP913X	HP-IB or fast HP-IB	0	0 thru 3	9
HP 9134A, Opt. 010 (h)	HP913X	HP-IB or fast HP-IB	0	0	9
HP 9134B	HP913X	HP-IB or fast HP-IB	0	0	—
HP 9134XV	HP913X	HP-IB or fast HP-IB	0	0	—
HP 9134XV, Opt. 010 (h)	HP913X	HP-IB or fast HP-IB	0	0	—
HP 9135 (f, all versions)	HP913X	HP-IB	2	0	4
HP 9135A (h)	HP913X	HP-IB or fast HP-IB	0	0 thru 3	9
HP 9135A, Opt. 010 (h)	HP913X	HP-IB or fast HP-IB	0	0	9
HP 9885M	HP9885	GPIO & DMA	—	0	1
HP 9885S	HP9885S	GPIO & DMA	—	1	1
HP 9895M (l)	HP9895	HP-IB or fast HP-IB	0	0	3
HP 9895M (r)	HP9895	HP-IB or fast HP-IB	0	1	3
HP 9895S (l)	HP9895	HP-IB or fast HP-IB	0	2	3
HP 9895S (r)	HP9895	HP-IB or fast HP-IB	0	3	3
HP 82901M (l)	HP82901	HP-IB	0	0	4
HP 82901M (r)	HP82901	HP-IB	0	1	4
HP 82901S (l)	HP82901	HP-IB	0	2	4
HP 82901S (r)	HP82901	HP-IB	0	3	4
HP 82902M	HP82902	HP-IB	0	0	4
HP 82902S	HP82902	HP-IB	0	2	4

(h) indicates hard-disc drive
(f) indicates flexible disc drive
(t) indicates tape drive
(l) indicates right drive (of dual-drive machine)
(r) indicates left drive (of dual-drive machine)
— indicates "not applicable"

Now let's take a closer look at each of these drives and briefly summarize the configuration considerations.

HP 7908, 7911, 7912, and 7914 Disc/Tape Drives
These machines all have a hard disc drive and tape (backup) drive; the differences between model numbers are primarily disc (and tape) capacities. Tape drives may be optionally deleted when ordering the machine. Using these drives requires CS80 and HPIB or FHPIB.

If the disc and tape drive have only one controller, both are located at the same primary address; this address has a random factory setting in the range 0 through 7. If disc and tape each have their own controller, each also has its own address setting. The address switch setting(s) are located on the rear panel below the HP-IB connector; see the installation manual for the switch definitions. The unit number of the disc is 0, and the unit number of the tape is 1.

To specify any of these drives, use the device type `CS80`. For instance, to specify the disc at interface select code 7 with primary address 0, use `:CS80,700`; to specify the tape on the same machine, use `:CS80,700,1`.

The default interleave factor for all CS80 discs is 1, which is optimal for an HP 7908 while using DMA. For the HP 7912, an interleave of 3 is optimal (while using DMA). Time-critical operations may perform better with an interleave of 3, 7, or 12.

Note that you should not use the COPY statement to do media backup between the disc drive and the tape drive. The utility program "TAPEBACKUP" has been provided with the BASIC Utilities Library for this purpose.

HP 9121S and 9121D Drives
The HP 9121S is a single-drive machine, and the HP 9121D is a dual-drive machine. Both use 3.5-inch flexible discs. Using these drives requires DISC and HPIB or FHPIB.

The factory default setting of the primary address is 0. The address switches are located on the rear panel; see the installation manual for switch definitions. With the HP 9121D, the left drive is set to unit 0, and the right drive is set to unit 1.

To specify these drives, use any of the device types `HP9121`, `HP8290X`, or `HP913X`. To specify the right drive on an HP 9121D at interface select code 7 with primary address 0, use `:HP9121,700,1`. To specify the drive of an HP 9121S at select code 8 with primary address 2, use `:HP9121,802`.

The default interleave factor is 2 for HP 9121 drives.

HP 9133, 9134, and 9135 Drives
All of these drives feature a 5.25-inch hard-disc drive. The HP 9133 contains an additional 3.5-inch flexible disc drive, which uses media compatible with HP 9121 drives. The HP 9135 contains an additional 5.25-inch flexible disc drive, which uses media compatible with internal and HP 8290X drives. Using these drives requires DISC and HPIB for flexible disc drives or FHPIB for other devices.

The drives are available in the following sizes and configurations:

Option	Size	Volumes
A	4 825 088	4
A, Option 010	4 825 088	1
B	9 681 920	1
V	4 825 058	1
V, Option 004	4 825 088	4
XV	14 522 880	1
XV Option 010	9 681 920	1

The available combinations are:

HP 9133	HP 9134	HP 9135	Options
Yes	Yes	Yes	A
Yes	Yes	Yes	A, Option 010
Yes	Yes	No	B
Yes	No	No	V
Yes	No	No	V, Option 004
Yes	Yes	No	XV
Yes	Yes	No	XV, Option 010

The factory default setting of the primary address switches for the hard disc on all three of these devices is 0; for the flexible discs (HP 9133 and HP 9135), the default address is 2. The switches are located on the rear panel; see the installation manual for switch definitions.

The default interleave factor for these drives is 2 (for flexible discs) and 9 (for hard discs). The interleave of the hard discs is 9 and cannot be changed.

HP 9885 Drives

The HP 9885 is a single-drive, single-sided, 8-inch, flexible disc drive. Unlike the other disc drives, using this device *requires* a GPIO Interface (HP 98622) and a DMA Interface (HP 98620). The factory default setting of the GPIO Interface' select code is 12; the range of allowable select codes is 8 through 31, as long as no other interface is set to the same select code. The DMA Interface has no select code setting. Using this drive requires HP9885 and GPIO.

The "INT LVL" switches are irrelevant when using the HP 9885. All "Option Select" switches (PCTL, PFLG, PSTS, HSHK, DIN, and DOUT) *must* be set to the "1" position (open). The "Data-In Clock Source" switches *must* be set as follows:

	Switch Name	Required Setting
LOWER	READ	1 (open)
	BSY	0 (closed)
	RDY	1 (open)
UPPER	READ	1 (open)
	BSY	0 (closed)
	RDY	1 (open)

There is no primary address setting for the HP 9885 drives, since these drives are connected to the computer through a GPIO Interface. The factory default settings for unit numbers depend on whether or not the drive is a master or slave drive; HP 9885M drives are masters, and HP 9885S drives are slaves. Master drives have a default unit number of 0, while the slaves have defaults of 1. Each master can control up to three slaves. The unit-number switch is a rotary switch located on the rear panel.

To specify an HP 9885 at interface select code 12 and with unit number 0, use `:HP9885,12`; to specify a drive at the same select code but with unit number 1, use `:HP9885,12,1`.

HP 9895 Drives
The HP 9895 disc drives are dual-drive machines which can use either single-sided or double-sided, 8-inch, flexible-disc media. The single-sided media are compatible with HP 9885 drive. Using these drives requires DISC and HPIB or FHPIB.

The primary address switch is located behind the removable front panel; the factory default address setting is 0. The unit number of the left drive is 0; for the right drive, it is 1.

To specify an HP 9895 drive, use the `HP9895` device type. For instance, to specify the right drive of an HP 9895 located at interface select code 7 with primary address 0, use `:HP9895,700,1`; to specify the left drive of the same machine, use `:HP9895,700`.

The default interleave factor for HP 9895 discs is 3.

HP 82901 and HP 82902 Drives
The HP 82901 drives are dual-drive machines, while the HP 82902 drives are single-drive machines. The M suffix (of the part number) denotes that the machine is a master drive; master drives can control one slave drive (either an HP 82901 or HP 82902 which has an S suffix). The media used with both of these 5.25-inch flexible disc drives are compatible with the internal and HP 9135 flexible disc drives. Using these drives requires DISC and HPIB

The primary address switches are located on the master drive's rear panel under the HP-IB connector. The factory default primary address setting is 0; see the installation manual for switch definitions. On the HP 82901M, the left drive is unit 0, and the right drive is unit 1; the single drive of the HP 82902M is unit 0. On an HP 82901S, the left drive is unit 2, and the right drive is unit 3; the single drive of an HP 82902S is unit 2.

To access these drives, use any of the `HP8290X` device types. For instance, to specify the left drive of an HP 82901 at select code 7 with primary address 0, use `HP82901,700`; to specify the right drive of the same machine, you could use `HP8290X,700,1`.

Non-Disc Mass Storage

Although mass storage is traditionally implemented using a magnetic surface such as a disc or drum, the protocols of file management can be applied to any device which stores data. Here is a summary.

- EPROM (Erasable, Programmable Read-Only Memory) cards. Although EPROMs store user-supplied data, they are much harder to write than to read. This, combined with their ruggedness and non-volatile storage, makes them well suited to read-only applications in environments too harsh for a disc. Because of their specialized nature, EPROMs are not discussed in this chapter on generalized mass storage. They are covered in their own chapter of *BASIC Interfacing Techniques*.
- Magnetic Bubble Memory cards. Like EPROMs, these devices provide non-volatile storage that is more rugged than a disc. Unlike EPROMs, bubble memory can be written to just as easily as it can be read.
- RAM Memory Volumes. Areas of the computer's RAM can be treated as though they were mass storage devices. Obviously, a RAM volume is volatile. However, they can be accessed faster than any other mass storage device. Some special tasks can benefit from this increased speed for intermediate operations and then copy the RAM volume to a non-volatile volume at the end of the job.

The next two sections describe some of the programming techniques needed to access bubble memory and RAM volumes.

Bubble memory

The Bubble Memory Card is very similar to an interface card. It has interface select code switches and installs in an accessory slot. The Installation Note for this card shows how to set the switches. Note that hardware interrupt level of 6 is recommended. The BUBBLE BIN is required to use this card.

The follow example show typical mass storage unit specifiers for a bubble memory card set to interface select code 30.

```
MASS STORAGE IS ":BUBBLE,30"
ASSIGN @File TO "data1:BUBBLE,30"
```

A bubble memory card always has only one unit (unit 0). This could be specified in the media specifier, such as `:BUBBLE,28,0`. However, zero is the default unit number in a media specifier, so there is really no point in specifying it.

All mass storage operations work with the bubble memory card. Just use the card's interface select code in the BUBBLE media specifier. There are *very rare* instances when hardware conflicts might occur during a bubble memory access. These pertain to hardware interrupt levels, which are described in *BASIC Interfacing Techniques*. Essentially, if an interrupt (at the same or higher priority) occurs during a bubble memory access, BASIC might not be able to service the bubble memory card quickly enough. If this happens, error 314 is reported. The cure is to lower the hardware interrupt level of the cards in conflict with the bubble card or restructure the program so that the conflicting operations (e.g. TRANSFER) do not occur at the same time.

When the bubble memory device was first initialized at the factory, any bad loops (memory locations) were logged and this information, called the Boot Loop, was stored in the bubble memory device itself. If you obtain repeated "read" errors or "buffer overflow" errors from the bubble memory card, try to re-initialize the volume. If the initialization fails, it is likely the Boot Loop information has been lost and the card should be returned to re-establish the Boot Loop.

RAM Volumes

Areas of the computers RAM may be treated as mass storage devices. These "memory volumes" or "RAM volumes" are volatile (all information is lost when the power goes off), but high speed. A typical use for RAM volumes is to copy a disc volume into memory, perform all necessary manipulations using the RAM volume, then copy the new information back to disc. Obviously, there are only certain applications which would benefit from this technique.

All mass storage operations work with RAM volumes. After they are created (described next), RAM volumes are accessed by using their unit number in a MEMORY media specifier. The following examples show typical mass storage unit specifiers for a RAM volume with unit number 7.

```
MASS STORAGE IS ":MEMORY,0,7"
ASSIGN @Ram TO "TEMP:MEMORY,0,7"
```

RAM volumes are created by the INITIALIZE statement. A special form of this statement is used, with a unit size parameter is the position normally occupied by the interleave factor. The device type is always `MEMORY`, and the device selector is always 0. Unit numbers 0 thru 31 may be used. Here are some examples.

```
INITIALIZE ":MEMORY,0,1",220
```

This creates a RAM volume that is 220 sectors long and is given unit number 1. Note that the unit size parameter is in 256-byte sectors, just like LIF file sizes.

If the unit size parameter is omitted, the result is a RAM volume that is the same size as a 5.25-inch or 3.5-inch disc. This a 1056 sectors, or 270 336 bytes. Although the default size RAM volume provides only 80 directory entries while the discs may contain up to 112 directory entries, if a disc is copied into the RAM volume, the entire directory will be copied.

The unit size of a RAM volume must be at least 4 sectors and can be as large as available memory permits. Two sectors are taken for system use, and about 1 sector of directory is created for each 100 sectors of unit size. The following table shows examples of size data for RAM volumes.

Specified Unit Size (sectors)	Total Overhead (sectors)	Maximum Number of Files in Directory
4	3	8
200	3	8
201	4	16
300	4	16
512	7	40
1000	11	72
1056	12	80
2000	21	152
2048	22	160

With BASIC 3.0 RAM volumes can be initialized while a program is running.

No RAM volumes exist at power-up or after a SCRATCH A. It is recommended that all BIN programs be loaded before RAM volumes are initialized. If a BIN program is loaded after a RAM volume is initialized, the memory used for the RAM volume cannot be recovered until the computer is turned off and back on again.

A RAM volume can be reinitialized to the same or different size. If the size is different, memory space may be lost until the next SCRATCH A.

Accessing Files

Before you can access a data file, you must assign an I/O path name to the file. Assigning an I/O path name to the file sets up a table in computer memory that contains various information describing the file, such as its type, which mass storage device it is stored on, and its location on the media. The I/O path name is then used in I/O statements (OUTPUT, ENTER, and TRANSFER) which move the data to and from the file. I/O path names are also used to transfer data to and from devices. Chapters 4, 5, 10, and 11 of *BASIC Interfacing Techniques* explain data transfers with devices and provide several relevant insights into data representations. However, in this chapter we deal only with I/O paths to files.

Every I/O path to a file maintains the following information:

Validity Flag — Tells whether the path is currently opened (assigned) or closed (not assigned).

Type of Resource — Holds the file type (ASCII or BDAT).

Device Selector — Stores the device selector of the drive. (I/O paths can also be associated with devices and buffers. See *BASIC Interfacing Techniques* for further details.)

Attributes — Such as FORMAT OFF and FORMAT ON.

File Pointer — There is a file pointer that points to the place in the file where the next data item will be read or written. The file pointer is updated whenever the file is accessed.

End-Of-File Pointer — An I/O path has an EOF pointer that points to the byte that follows the last byte of the file.

Opening an I/O Path

I/O path names are similar to other variable names, except that I/O path names are preceded by the "@" character. When an I/O path name is used in a statement, the system looks up the contents of the I/O path name and uses them as required by the situation.

To open an I/O path to a file (to set the validity flag to Open), assign the I/O path name to a file specifier by using an ASSIGN statement. For example, executing the following statement:

```
ASSIGN @Path1 TO "Example"
```

assigns an I/O path name called @Path1 to the file "Example". The file that you open must already exist and must be a data file. If the file does not satisfy one of these requirements, the system will return an error. If you do not use an msus in the file specifier, the system will look for the file on the current MASS STORAGE IS device. If you want to access a different device, use the msus syntax described earlier. For instance, the statement:

```
ASSIGN @Path2 TO "Example:HP9895,707"
```

opens an I/O path to the file "Example" on an HP 9895 disc drive, interface select code 7 and primary address 7. You must include the protect code if the file has one.

ASSIGNing an I/O path name to a file has the following effect on the I/O path table:

- If the I/O path is currently open, the system *closes* the I/O path and then *re-opens* it. If the I/O path is not currently open, it is opened. In both cases, the system sets the validity flag to Open.
- The file type (ASCII or BDAT) and its msus are recorded.
- The specified attributes are assigned to the I/O path name. If an attribute is not specified, the appropriate default attribute is assigned.
- The file pointer is positioned to the beginning of the file.
- If the path name is associated with a BDAT file, the EOF pointer from the system sector is copied to the I/O path table.

Once an I/O path has been opened to a file, you always use the path name to access the file. An I/O path name is only valid in the context in which it is opened, unless you pass it as a parameter or put it in the COM area. To place a path name in the COM area, simply specify the path name in a COM statement before you ASSIGN it. For instance the two statements below would declare an I/O path name in an unnamed COM area and then open it:

```
100   COM @Path3
110   ASSIGN @Path3 TO "File1"
```

Assigning Attributes

When you open an I/O path, certain attributes are assigned to it which define the way data is to be read and written. There are two attributes which control how data items are represented: FORMAT ON and FORMAT OFF. With FORMAT ON, ASCII data representations are used; with FORMAT OFF, the system's internal data representations are used. Additional attributes are available, which provide control of such functions as parity generation and checking, converting characters, and changing end-of-line (EOL) sequences. See the *BASIC Language Reference* or Chapter 10 of *BASIC Interfacing Techniques* for further details.

As mentioned in the tutorial section, BDAT files can use either data representation; however, ASCII files only permit ASCII-file format. Therefore, if you specify FORMAT OFF for an I/O path to an ASCII file, the system ignores it. The following two examples of ASSIGN statements specify a FORMAT attribute.

```
ASSIGN @Path1 TO "File1";FORMAT OFF
```

If "File1" is a BDAT file, the FORMAT OFF attribute specifies that the internal data formats are to be used when sending and receiving data through the I/O path. If the file is of type ASCII, the attribute will be ignored. *Note that FORMAT OFF is the default FORMAT attribute for BDAT files.*

Executing the following statement directs the system to use the ASCII data representation (if possible) when sending and receiving data through the I/O path.

```
ASSIGN @Path2 TO "File2";FORMAT ON
```

If "File2" is a BDAT file, data will be written using ASCII format, and data read from it will be interpreted as being in ASCII format. For an ASCII file, this attribute is redundant since ASCII-file format is the only data representation allowed anyway.

If you want to change the attribute of an I/O path, you can do so by specifying the I/O path name and attribute in an ASSIGN statement while excluding the file specifier. For instance, if you wanted to change the attribute of @Path2 to FORMAT OFF, you could execute:

```
ASSIGN @Path2;FORMAT OFF
```

Alternatively, you could re-enter the entire statement:

```
ASSIGN @Path2 TO "File2";FORMAT OFF
```

These two statements, however, are not identical. The first one only changes the FORMAT attribute. The second statement resets the entire I/O path table (e.g., resets the file pointer to the beginning of the file).

It is important to note that once a file is written, changing the FORMAT attribute of an I/O path to the file should only be attempted by experienced programmers. *In general, data should always be read in the same manner as it was written.* For instance, data written to a BDAT file with FORMAT OFF should also be read with FORMAT OFF, and vice versa. In addition, the same data types should be used to write the file as to read the file. For instance, if data items were written as INTEGERs, they should also be read as INTEGERs.

In theory, there is no limit to the number of I/O paths you can ASSIGN to the same file. Each I/O path, however, has its own file pointer and EOF pointer, so that in practice it can become exceedingly difficult to keep track of where you are in a file if you use more than one I/O path. *We recommend that you use only one I/O path for each file.*

Closing I/O Paths

I/O path names not in the COM area are closed whenever the system moves into a stopped state (e.g., STOP, END, SCRATCH, EDIT, etc.). I/O path names local to a context are closed when control is returned to the calling context. Re-ASSIGNing an I/O path name will also cancel its previous association.

You can also explicitly cancel an I/O path by ASSIGNing the path name to an * (asterisk). For instance, the statement:

```
ASSIGN @Path2 TO *
```

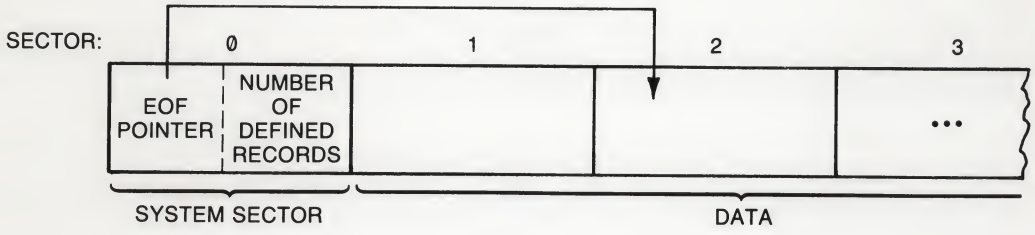
closes @Path2 (sets the validity flag to Closed). @Path2 cannot be used again until it is Re-ASSIGNed. You can Re-ASSIGN a path name to the same file or to a different file.

Reading and Writing BDAT Files

There are many alternatives for storing and retrieving data when using BDAT files. You can choose internal ASCII or user-defined formats and serial or random access. The following descriptions of OUTPUTing and ENTERing data apply only to BDAT files using FORMAT OFF. For more information about OUTPUT, ENTER, IMAGE specifiers, and additional attributes, see the *BASIC Interfacing Techniques* manual.

System Sector

On the disc, every BDAT file is preceded by a system sector that contains an End-Of-File pointer and the number of defined records in the file. All data is placed in succeeding sectors. You cannot directly access the system sector. However, as you shall see later, it is possible to indirectly change the value of an EOF pointer.



Defined Records

To access a BDAT file randomly, you specify a particular defined record. Records are the smallest units in a file directly addressable by a random OUTPUT or ENTER. They can be anywhere from 1 through 65 534 bytes long. Both the length of the file and the length of the defined records in it are specified when you create the file. For example, the statement:

```
CREATE BDAT "Example",7,128
```

would create a file called "Example" with 7 defined records, each record being 128 bytes long. If you don't specify a record length in the CREATE BDAT statement, the system will set each record to the default length of 256 bytes.

Both the record length and the number of records are rounded to the nearest integer. Further, the record length is rounded up to the nearest even integer. For example, the statement:

```
CREATE BDAT "Odd",3.5,28.7
```

would create a file with 4 records, each 30 bytes long. On the other hand, the statement:

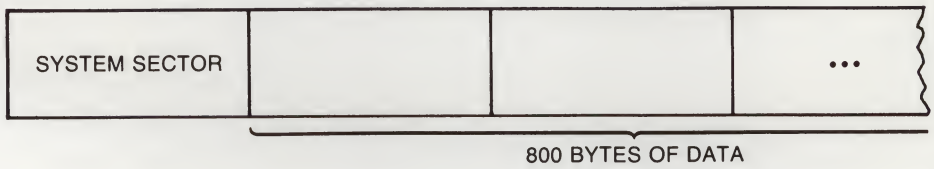
```
CREATE BDAT"Odder",3.49,28.3
```

would create a file with three records, each 28 bytes long.

Once a file is created, you cannot change its length or the length of its records. You must therefore calculate the record size and file size required before you create a file.

Choosing A Record Length

The most important consideration in selecting of a proper record length is the type of data being stored and the way you want to retrieve it. Suppose, for instance, that you want to store 100 real numbers in a file, and be able to access each number individually. Since each REAL number uses 8 bytes, the data itself will take up 800 bytes of storage.



The question is how to divide this data into records. If you define the record length to be 8 bytes, then each REAL number will fill a record. To access the 15th number, you would specify the 15th record. If the data is organized so that you are always accessing two data items at a time, you would want to set the record length to 16 bytes.

The worst thing you can do with data of this type is to define a record length that is not evenly divisible by eight. If, for example, you set the record length to four, you would only be able to randomly access half of each real number at a time. In fact, the system will return an End-Of-Record condition if you try to randomly read data into REAL variables from records that are less than 8 bytes long.

So far, we have been talking about a file that contains only REAL numbers. For files that contain only INTEGERs, you would want to define the record length to be a multiple of two. To access each INTEGER individually, you would use a record length of two; to access two INTEGERs at a time, you would use a record length of four, and so on.

Files that contain string data present a slightly more difficult situation since strings can be of variable length. If you have three strings in a row that are 5, 12, and 18 bytes long, respectively, there is no record length less than 22 that will permit you to randomly access each string. If you select a record length of 10, for instance, you will be able to randomly access the first string but not the second and third.

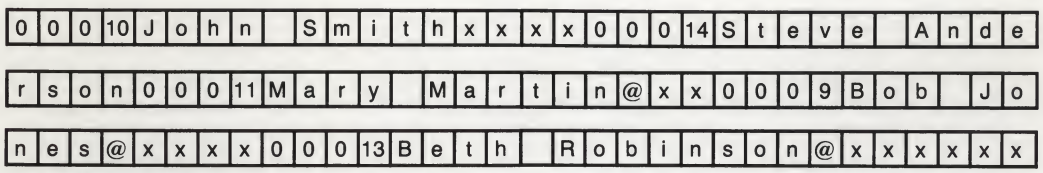
If you want to access strings randomly, therefore, you should make your records long enough to hold the largest string. Once you've done this, there are two ways to write string data to a BDAT file. The first, and easiest, is to output each string in random mode. In other words, select a record length that will hold the longest string and then write each string into its own record. Suppose, for example, that you wanted to OUTPUT the following 5 names into a BDAT file and be able to access each one individually by specifying a record number.

John Smith
Steve Anderson
Mary Martin
Bob Jones
Beth Robinson

The longest name, "Steve Anderson", is 14 characters. To store it in a BDAT file would require 18 bytes (four bytes for the length header). So you could create a file with record length of 18 and then OUTPUT each item into a different record:

```
100   CREATE BDAT "Names",5,18          ! Create a file.
110   ASSIGN @File TO "Names"           ! Open an I/O path
120   OUTPUT @File,1;"John Smith"       ! Write names to
130   OUTPUT @File,2;"Steve Anderson"   !  successive records
140   OUTPUT @File,3;"Mary Martin"      !  in file
150   OUTPUT @File,4;"Bob Jones"
160   OUTPUT @File,5;"Beth Robinson"
```

On the disc, the file "Names" would look like the figure below. The four-byte length headers show the decimal value of the bytes in the header. The data are shown as ASCII characters.



1 = length header
x = whatever data previously resided in that space
@ = pad character

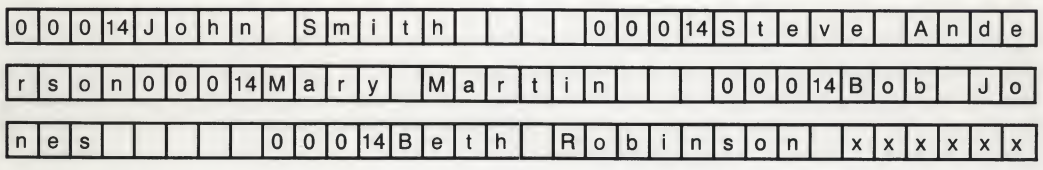
The unused portions of each record contain whatever data previously occupied that physical space on the disc.

The other method for writing strings to a BDAT file is to pad each entry so that they are all of uniform length. While this method involves more programming, it allows you to pad the unused portions of each record with whatever characters you choose. It also permits you to read and write the data serially as well as randomly. The program below shows how you might enter the five names into a file by padding each name with spaces.

```
100   CREATE BDAT "Names",5,18      ! Create file.
110   ASSIGN @Path1 TO "Names"      ! Open I/O path to file.
120   FOR Entry=1 TO 5
130      LINPUT Name$[1;14]         ! Get names from keyboard
140      OUTPUT @Path1;Name$        ! Write name to file
150   NEXT Entry
```

In this program, we input each name from the keyboard and so that it is padded with spaces to a length of 14 bytes. With the length header, each entry is 18 bytes, or one record. In line 140, we write the name serially to the file. Since every data item is 18 bytes, there is no need to write randomly, although we could if we wanted to. Since the LINPUT statement is limited to 14 bytes, any names that are longer than 14 characters are automatically truncated.

If we had used the second program to enter the names, file "Names" would look like the figure below:



EOF Pointers

There are two types of End-Of-File pointers. One is maintained internally in the I/O path table and the other resides in the system sector. The two pointers are always updated at the same time so that they always agree with one another. (This may not be true if you use more than one I/O path to OUTPUT data to one file.) The two pointers are updated when either of the two conditions below occur.

- If, after an OUTPUT statement has been executed, the file pointer value is greater than the EOF pointers, the EOF pointers are moved to the file pointer position.
- If an OUTPUT statement contains the "END" secondary word, the EOF pointers are moved to the file pointer position regardless of their current values.

The function of EOF pointers is to mark the logical end of a data file. Every file also has a physical EOF — the last byte reserved for the file when you create it. The EOF pointers cannot point beyond the physical EOF. The EOF pointer marks the point at which no more data can be read. Also, you cannot randomly write data more than one record past the EOF position.

If you have a 100-record file, and the EOF pointers point to the 50th record, records 50 through 100 cannot be read. If you attempt to read data beyond an EOF, an EOF condition occurs. EOF conditions can be trapped with an ON END statement. If you do not trap it, an EOF condition will cause Error 59. Attempting to read or write beyond the physical EOF will also result in an EOF condition. EOF conditions are described in more detail later in this chapter.

Moving EOF Pointers

When you first create a file, the EOF pointer in the system sector points to the first byte in the file. When you ASSIGN an I/O path to a file, the pointer in the system sector is copied to the I/O path table. As you OUTPUT data items to the file, both EOF pointers are changed so that they point to the next byte. This is also where the file pointer is positioned.

If you overwrite a file, however, the EOF pointers will not necessarily agree with the file pointer. For example, suppose you write 100 bytes to a file, and then re-ASSIGN the I/O path. By re-ASSIGNing, you move the file pointer back to the first byte in the file. The EOF markers, though, still point to the 101st byte. They will not be changed until the file pointer value is greater than 101, or until you specify an "END" in an OUTPUT statement.

The secondary word "END" is used to move the EOF pointers backwards. It forces the EOF pointers to be re-positioned to the file pointer byte, even if the new position is "earlier" in the file than their current position. In effect, this "shrinks" the file, causing data that lies past the new EOF position to become inaccessible.

Writing Data

Data is always written to a file with an OUTPUT statement via an I/O path. You can OUTPUT numeric and string variables, numeric and string expressions, and arrays. When you OUTPUT data with the FORMAT OFF, data items are written to the file in internal format (described earlier).

There is no limit to the number of data items you can write in a single OUTPUT statement, except that program statements are limited to two CRT lines. Also, if you try to OUTPUT more data than the file can hold, or the record can hold (if you are using random access), the system will return an EOF or EOR condition. If an EOF or EOR condition occurs, the file retains any data output ahead of the end condition.

There is also no restriction on mixing different types of data in a single OUTPUT statement. The system decides which data type each item is before it writes the item to the disc. Any item enclosed in quotes is a string. Numeric variables and expressions are OUTPUT according to their type (8 bytes for REALs and 2 bytes for INTEGERs). Arrays are written to the file in row major order (rightmost subscript varies quickest).

Each data item in an OUTPUT statement should be separated by either a comma or semi-colon (there is no operational difference between the two separators with FORMAT OFF). Punctuation at the end of an OUTPUT statement is ignored with FORMAT OFF.

Serial OUTPUT

Data is written serially to BDAT files whenever you do not specify a record number in an OUTPUT statement. When data is written serially, each data item is stored immediately after the previous item without any type of separator. Sector and record boundaries are ignored. Data items are written to the file one by one, starting at the current position of the file pointer. As each item is written, the file pointer is moved to the next byte. After all of the data items have been OUTPUT, the file pointer points to the first byte following the last byte just written.

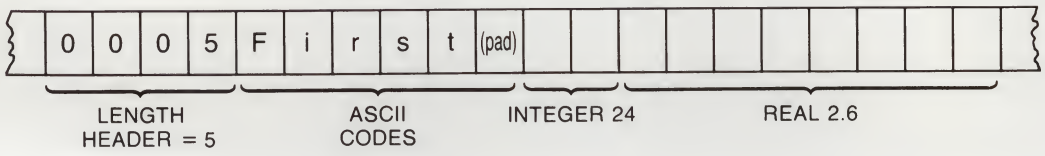
There are a number of circumstances where it is faster and easier to use serial access instead of random access. The most obvious case is when you want to access the entire file at once. If, for example, you have a list of data items that you want to store in a file and you know that you will never want to read any of the items individually, you should write the data serially. The fastest way to write data serially is to place the data in an array and then OUTPUT the entire array at once.

Another situation where you might want to use serial access is if the file is so small that it can fit entirely into internal memory at once. In this case, even if you want to change individual items, it might be easier to treat the entire file as one or more arrays, manipulate as desired, and then write the entire array(s) back to the file.

The examples below illustrate how data is stored serially in a file. The statement:

```
OUTPUT @Path1;"First",24;2.6,
```

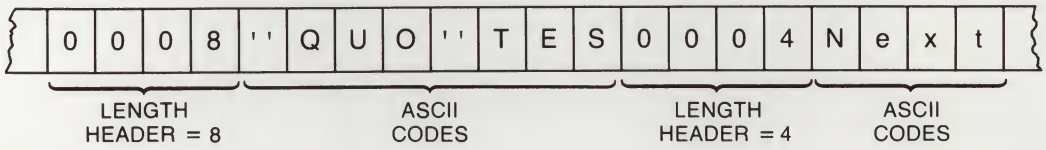
would result in the following storage format:



Note that quotation marks around a string are not written to the file. To write quote marks to a file, enter two quote marks for every one you want to OUTPUT. Note also that separators are not written to the file. To write a comma or semi-colon to a file, you must enclose it in quotes. For instance, the statement:

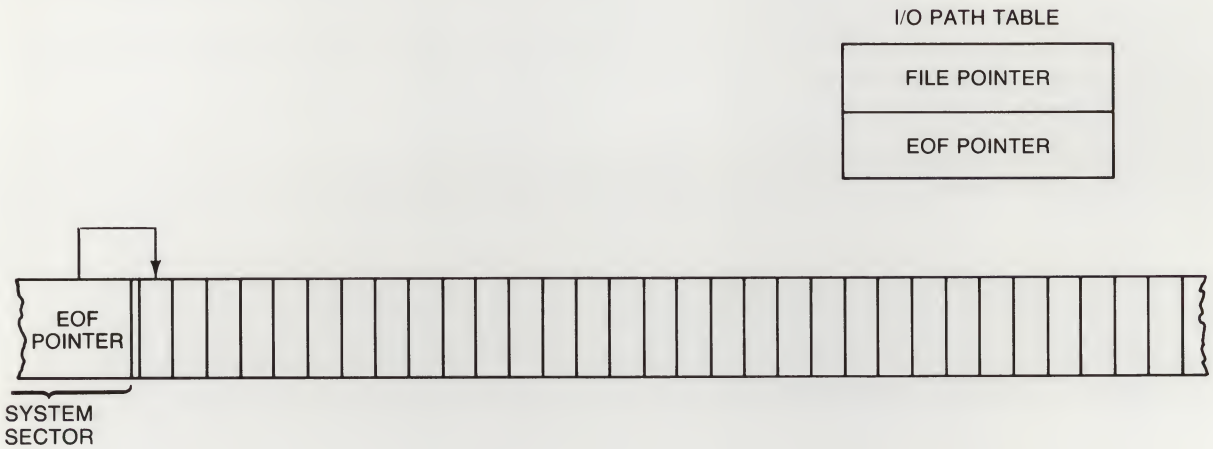
```
OUTPUT @Path1; """QUO""TES","Next"
```

would be stored:



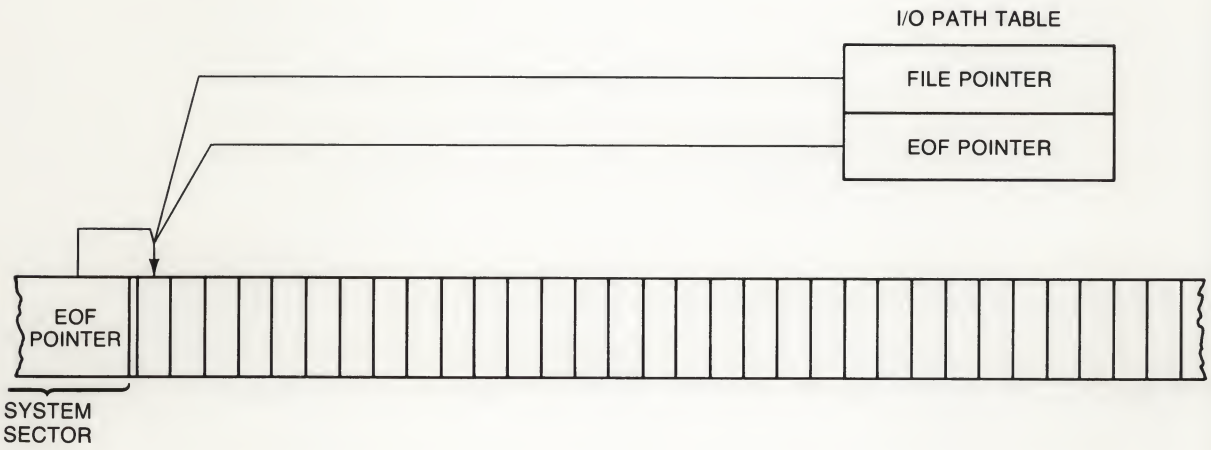
The following sequence of serial OUTPUT statements show how data is written to a BDAT file and how the file pointer and EOF pointers are updated.

```
CREATE BDAT "Example",4,128
```



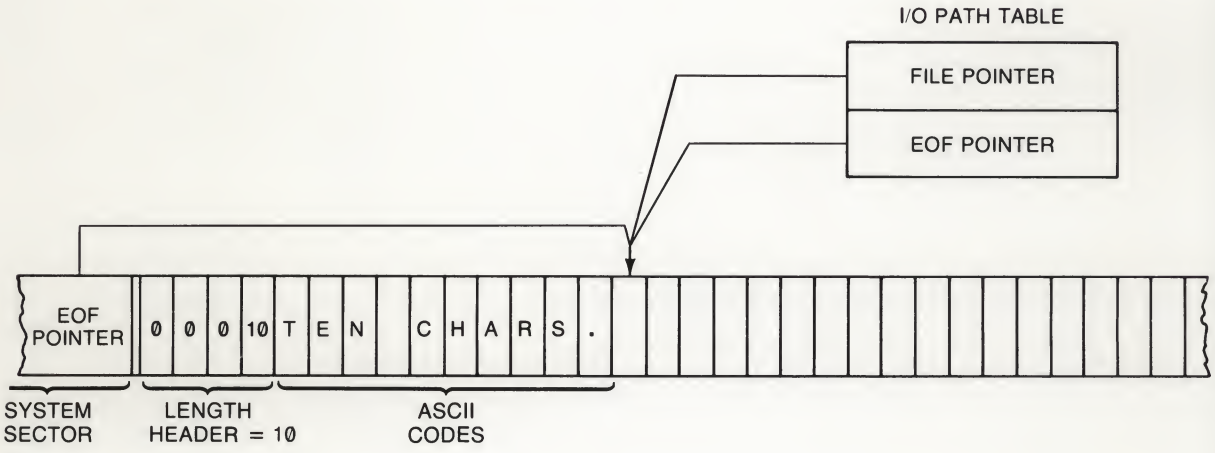
Creates a BDAT file with four 128-byte records. The EOF pointer in the system sector points to the first byte in the file.

```
ASSIGN @Path1 TO "Example"
```



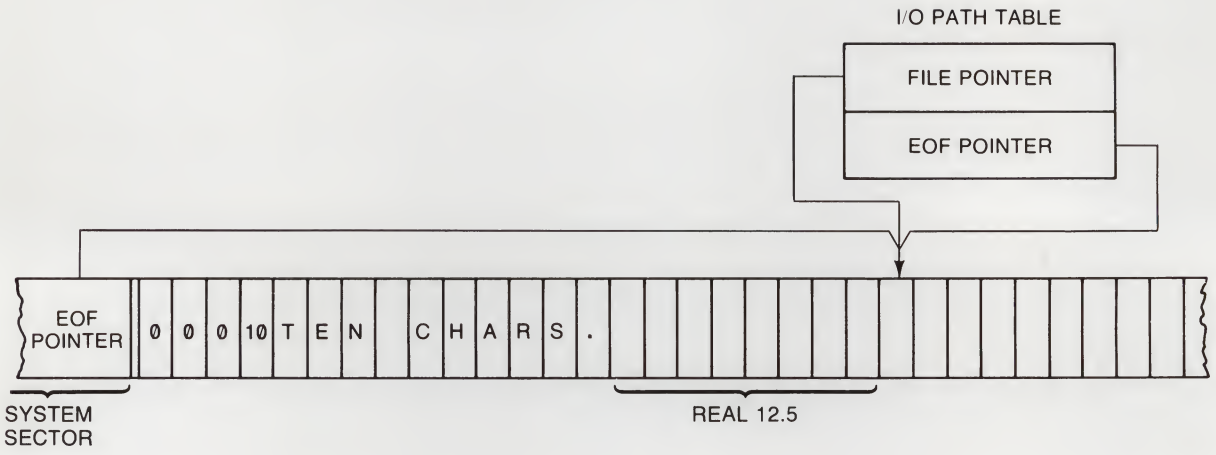
Opens an I/O path to "Example". The EOF marker in the system sector is copied to the I/O path table. The file pointer is positioned to the beginning of the file.

```
OUTPUT @Path1;"TEN CHARS."
```



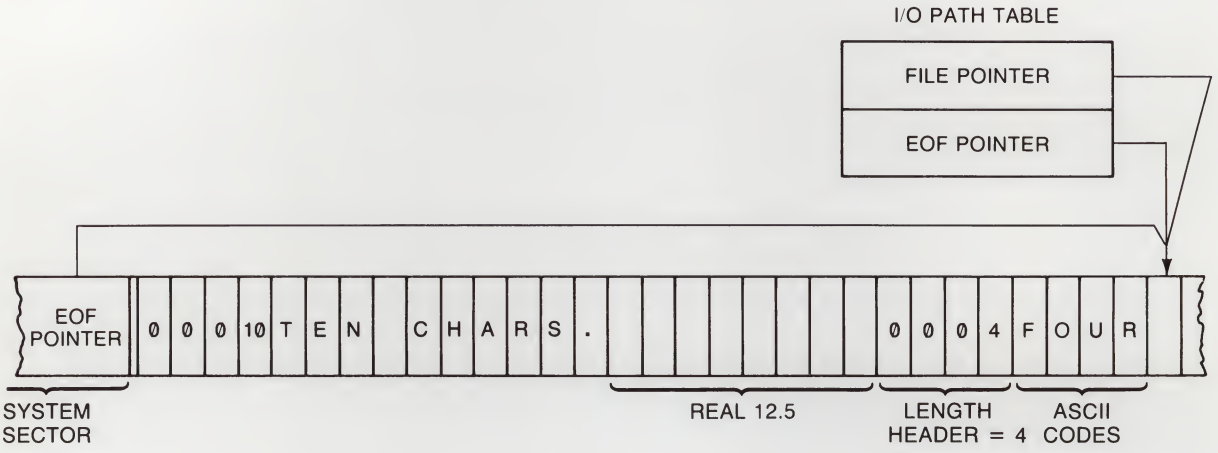
Fourteen bytes are written to the file. The EOF pointers are moved to the 15th byte. The file pointer also points to the 15th byte.

```
OUTPUT @Path1;12.5,END
```



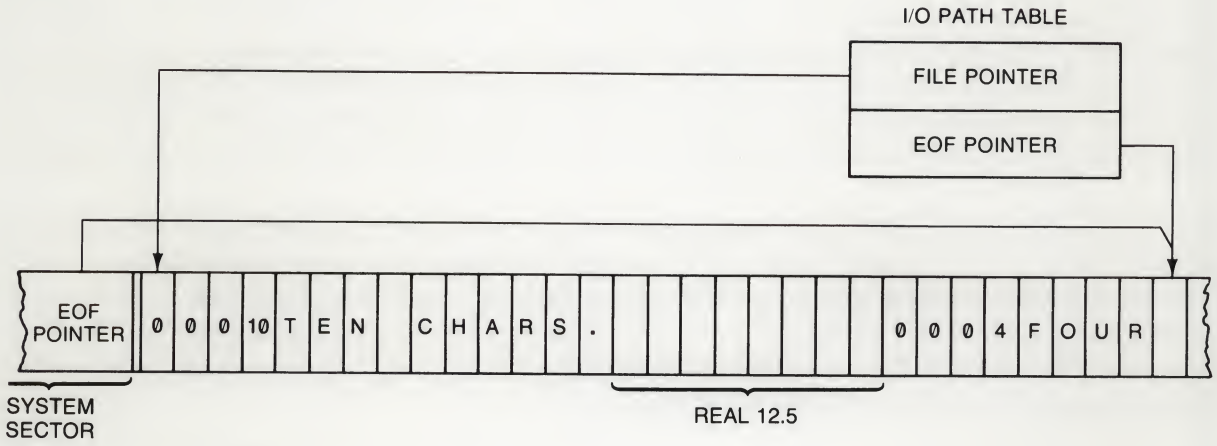
Eight more bytes are written to the file. The file pointer now points to the 23rd byte. Both the EOF in the I/O path table and the EOF in the system sector are updated to 23.

```
OUTPUT @Path1;"FOUR"
```



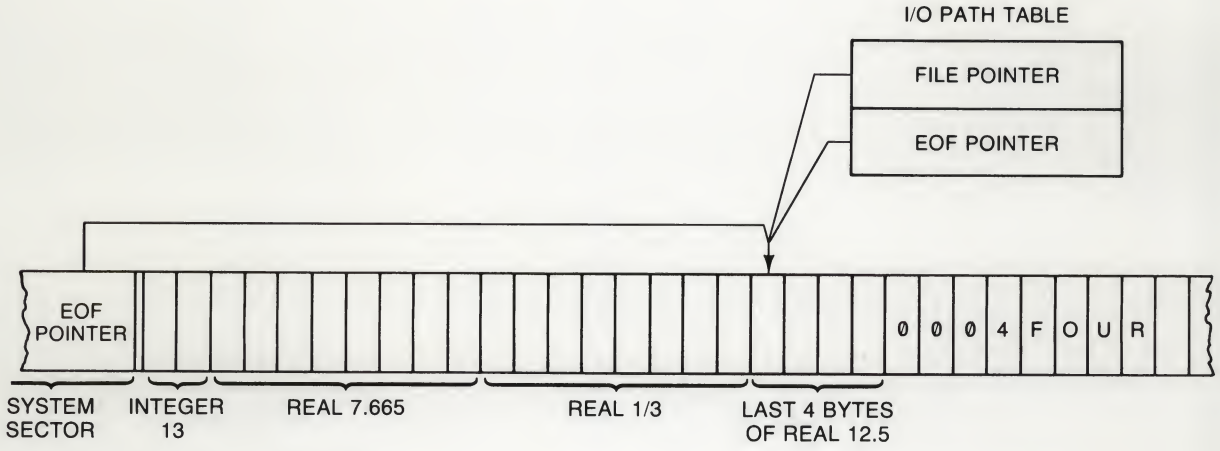
Eight more bytes are written to the file. The file pointer now points to the 31st byte. The EOF markers are updated to 31 because 31 is greater than 23, the current EOF value.

```
ASSIGN @Path1 TO "Example"
```



Re-ASSIGNs the I/O path name. The file pointer is positioned back to the beginning of the file. The system sector EOF value is copied to the I/O path table.

```
OUTPUT @Path1;13,7.665,1/3,END
```



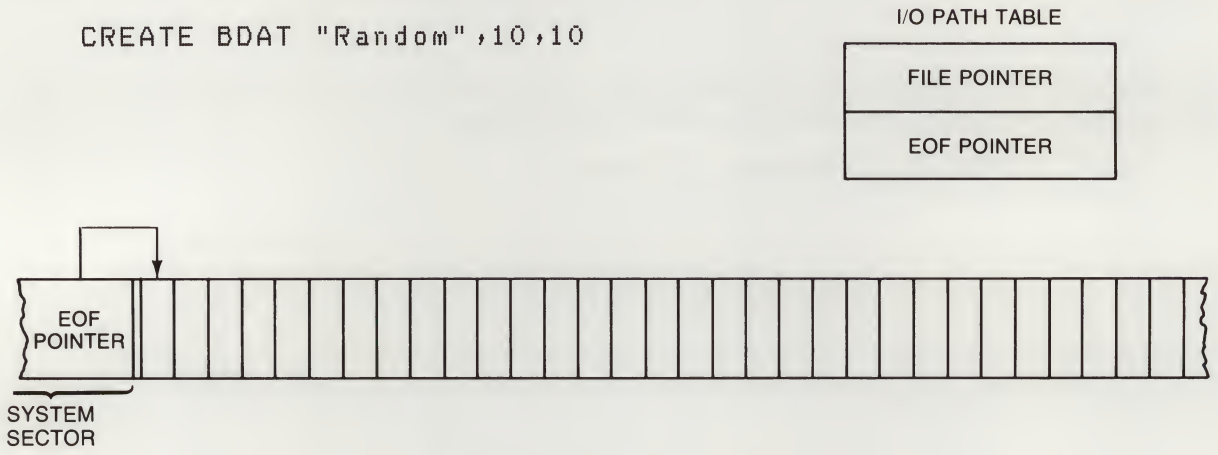
18 bytes (one INTEGER and two REALs) are OUTPUT, starting at the beginning of the file. The original data, therefore, is overwritten. The file pointer points to the 19th byte. The EOF markers are also positioned to 19 because the statement contains the "END" secondary word.

Random OUTPUT

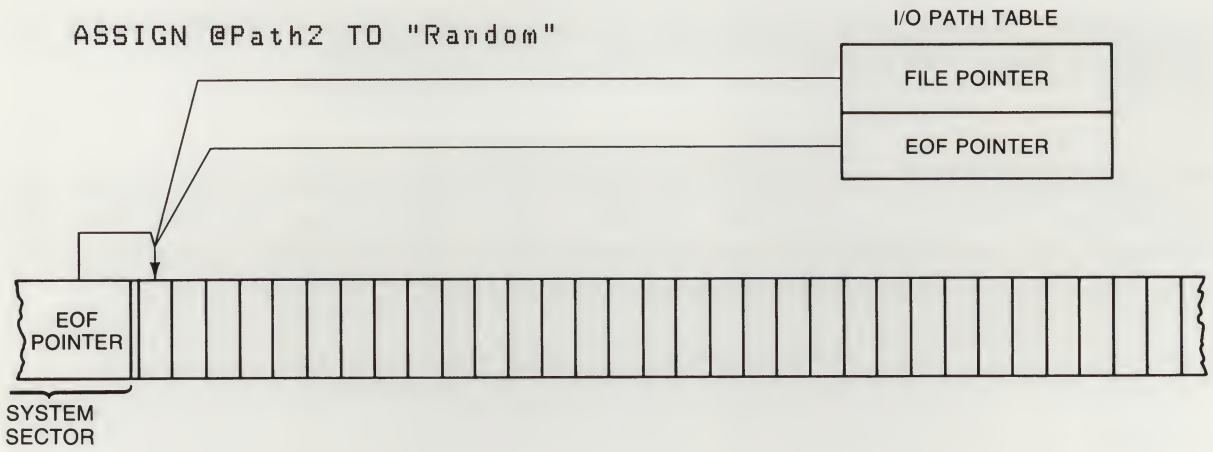
Random OUTPUT allows you to write to one record at a time. As with serial OUTPUT, there are EOF and file pointers that are updated after every OUTPUT. The EOF pointers follow the same rules as in serial access. The file pointer positioning is also the same, except that it is moved to the beginning of the specified record before the data is OUTPUT. If you wish to write randomly to a newly created file, either use a CONTROL statement to position the EOF in the last record, or start at the beginning of the file and write some "dummy" data into every record.

If you attempt to write more data to a record than the record will hold, the system will return an End-Of-Record (EOR) condition. An EOF condition will result if you try to write data more than one record past the EOF position. EOR conditions are treated by the system just like EOF conditions, except that they return Error 60 instead of 59 if they are not trapped by ON END. Data already written to the file before an EOR condition arises will remain intact. The examples below illustrate how data is stored randomly.

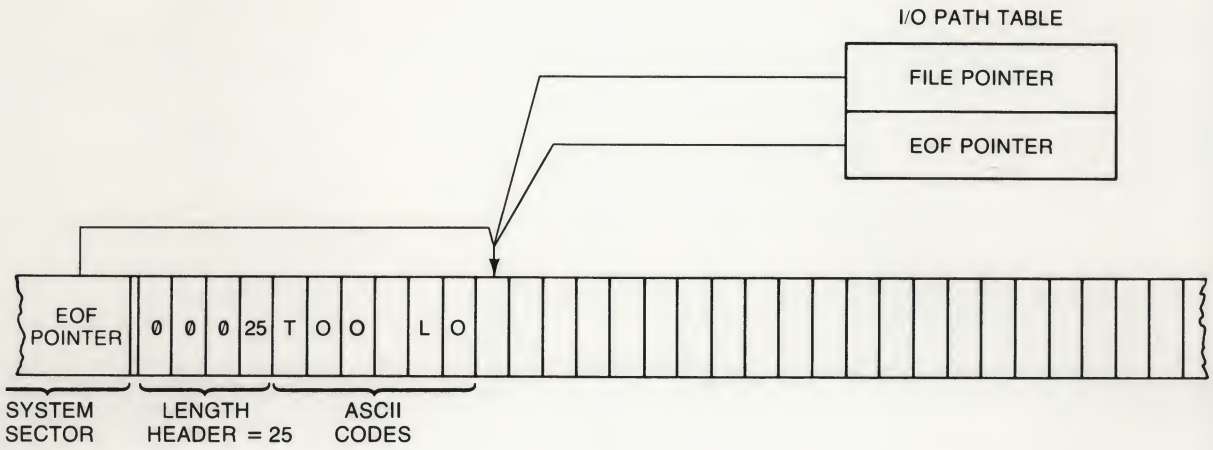
```
CREATE BDAT "Random",10,10
```




```
ASSIGN @Path2 TO "Random"
```

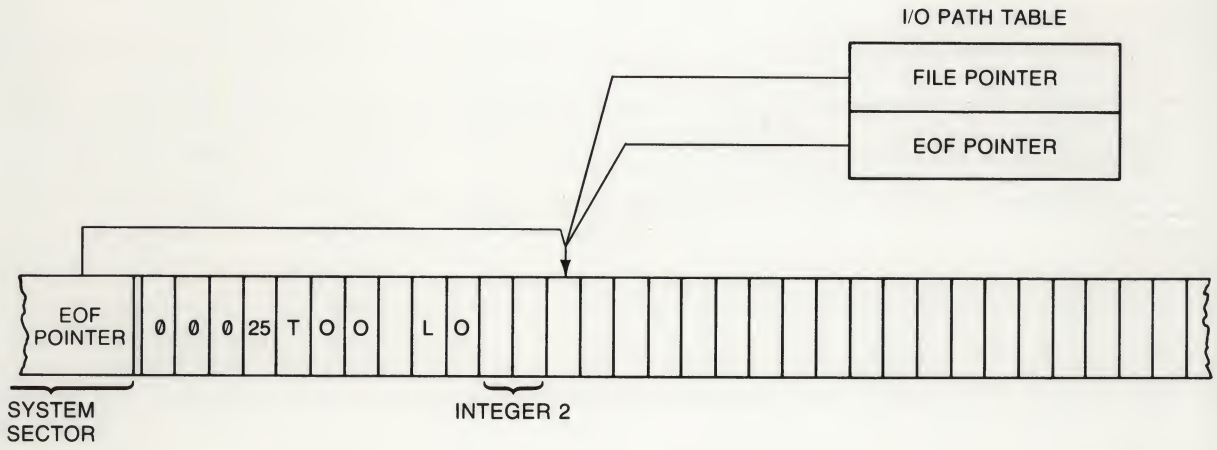


```
OUTPUT @Path2,1;"TOO LONG TO FIT IN RECORD"
```

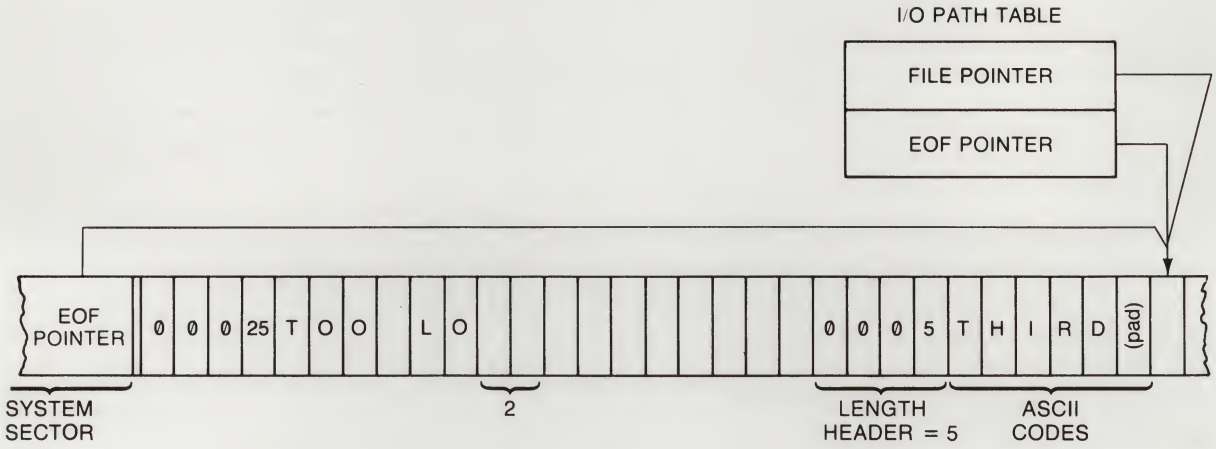


Even though this statement produces an EOR condition, the EOF markers and file pointer are still updated. The ON END statement can be used to trap their error. Also, the length header represents the length of the string characters sent to the file, since the whole string is not written out.

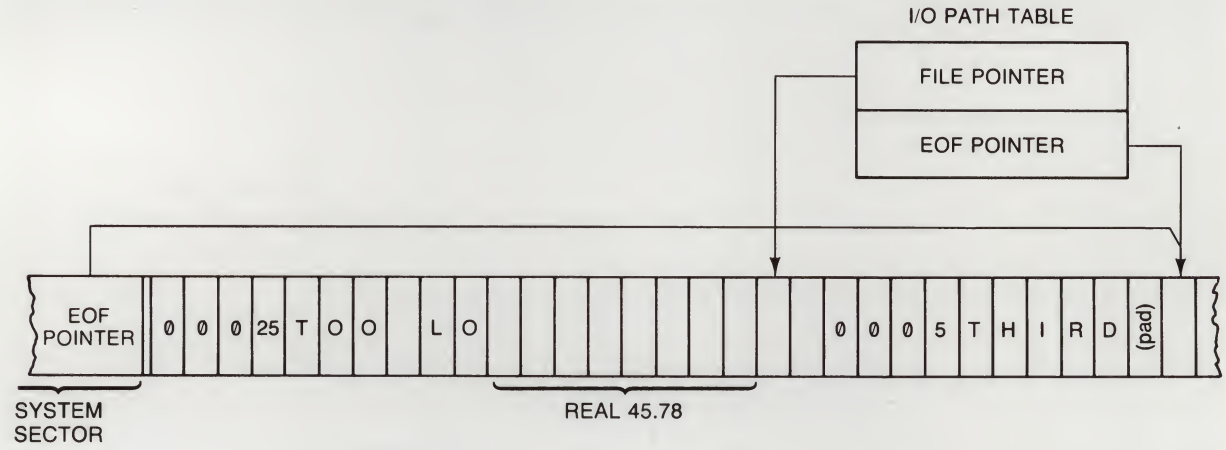
```
OUTPUT @Path2,2;2
```



```
OUTPUT @Path2,3;"THIRD"
```




```
OUTPUT @Path1,2;45.78
```



Reading Data From BDAT Files

Data is read from files with the ENTER statement. As with OUTPUT, data is passed along an I/O path. You can use the same I/O path you used to OUTPUT the data or you can use a different I/O path.

You can have several variables in a single ENTER statement. Each variable must be separated by either a comma or semi-colon. It is extremely important to make sure that your variable types agree with the data types in the file. If you wrote a REAL number to a file, you should ENTER it into a REAL variable; INTEGERs should be entered into INTEGER variables; and strings into string variables. The rule to remember is: "Read it the way you wrote it."

When reading data into a string variable, it is important to remember that the system will interpret the first four bytes after the file pointer as a length header. It will then try to ENTER as many characters as the length header indicates. If the string has been padded by the system to make its length even, the pad character is not read into the variable.

After an ENTER statement has been executed, the file pointer is positioned to the next unread byte. If the last data item was a padded string, the file pointer is positioned after the pad. If you use the same I/O path name to read and write data to a file, the file pointer will be updated after every ENTER and OUTPUT statement. If you use different I/O path names, each will have its own file pointer which is independent of the other. However, be aware that each also has its own EOF pointer and that these pointers may not match, which causes problems.

Entering data does not affect the EOF pointers. However, you cannot read data at or beyond the byte marked by the EOF pointers. If you attempt to read past an EOF marker, the system will return an EOF condition.

In addition to making sure that data types agree, it is also advisable to make sure that access modes agree. If you wrote data serially, you should read it serially; and if you wrote it randomly, you should read it randomly. There are a few exceptions to this rule which we discuss later. However, you should be aware that mixing access modes will often lead to erroneous results unless you are aware of the precise mechanics of the file system.

Serial ENTER

When you read data serially, the system enters data into variables starting at the current position of the file pointer and proceeds byte by byte until all of the variables in the ENTER statement have been filled. If there is not enough data in the file to fill all of the variables, the system returns an EOF condition. All variables that have already taken values before the condition occurs retain their values.

In the program below, we OUTPUT five data items serially, and then retrieve the data items with a serial ENTER statement.

```
10  CREATE BDAT "STORAGE",1
20  ASSIGN @Path TO "STORAGE"
30  INTEGER Num,First,Fourth
40  Num=5
60  OUTPUT @Path;Num,"squared"," equals",Num*Num,".",END
70  ASSIGN @Path TO "STORAGE"
80  ENTER @Path;First,Second$,Third$,Fourth,Fifth$
90  PRINT First;Second$;Third$,Fourth,Fifth$
100 END

5 squared equals 25.
```

Note that we re-ASSIGNed the I/O path in line 70. This was done to re-position the file pointer to the beginning of the file. If we had omitted this statement, the ENTER would have produced an EOF condition. Note also that the OUTPUT statement includes END, which specifies that the EOF pointer is to be moved to match the file pointer at statement completion. In this case, the END is redundant.

Random ENTER

When you ENTER data in random mode, the system starts reading data at the beginning of the specified record and continues reading until either all of the variables are filled or the system reaches the EOR or EOF. If the system comes to the end of the record before it has filled all of the variables, an EOR condition is returned.

In the following example, we randomly OUTPUT data to 10 successive records, and then ENTER the data into an array in reverse order.

```
10      CREATE BDAT "SQ_ROOTS,5,2*8
20      ASSIGN @Path TO "SQ_ROOTS"
30      FOR Inc=1 to 5
40         OUTPUT @Path,Inc;Inc,SQR(Inc)
50      NEXT Inc
60      FOR Inc=5 TO 1 STEP -1
70         ENTER @Path,Inc;Num(Inc),Sqroot(Inc)
80      NEXT Inc
90      PRINT "Number","Square Root"
100     FOR Inc=1 TO 5
110        PRINT Num(Inc),Sqroot(Inc)
120     NEXT Inc
130     END

Number      Square Root
1           1
2           1.41421356237
3           1.73205080757
4           2
5           2.2360679775
```

In this example, there was no need to re-ASSIGN the I/O path because the random ENTER automatically re-positions the file pointer.

Executing a random ENTER without a variable list has the effect of moving the file pointer to the beginning of the specified record. This is useful if you want to serially access some data in the middle of a file. Suppose, for instance, that you have a file containing 100 8-byte records, and each record has a REAL number in it. If you want to read the last 50 data items, you can position the file pointer to the 51st record and then serially read the remainder of the file into an array.

```
.
.
.
100     REAL Array(1:50)
110     ENTER @Realpath,51
120     ENTER @Realpath;Array(*)
.
.
.
```

Single-Byte Access

You can define records to be just one byte long. In this case, it doesn't make sense to read or write one record at a time since even the shortest data type require two bytes to store a number.

Random access to one-byte records, therefore, has its own set of rules. When you access a one-byte record, the file pointer is positioned to the specified byte. From there, the access proceeds in serial mode. Random OUTPUTs write as many bytes as the data item requires, and random ENTERs read enough bytes to fill the variable. The example below illustrates how you can read and write randomly to one-byte records.

```
10      INTEGER Int
20      CREATE BDAT "BYTE",100,1
30      ASSIGN @Bytepath TO "BYTE"
40      OUTPUT @Bytepath,1;3.67
50      OUTPUT @Bytepath,9;3
60      OUTPUT @Bytepath,11;"string"
70      ENTER @Bytepath,9;Int
80      ENTER @Bytepath,1;Real
90      ENTER @Bytepath,11;Str$
100     PRINT Real
110     PRINT Int
120     PRINT Str$
130     END

3.67
3
string
```

Note that we had to declare the variable "Int" as an INTEGER. If we hadn't, the system would have given it the default type of REAL and would therefore have required 8 bytes (also 8 records).

General Mass Storage Operations

Trapping EOF and EOR Conditions

An EOF condition exists whenever the system attempts to read data at, or beyond, the byte marked by the EOF pointers. The EOR condition will arise if you attempt to randomly read or write beyond the particular record specified. If, for example, you try to randomly OUTPUT a 20-character string into a 10-byte record, an EOR condition will occur. EOF conditions will also result whenever you try to read or write beyond the physical end-of-file.

EOF and EOR conditions can be trapped with an ON END statement. ON END is similar to ON ERROR except that it only traps EOF/EOR conditions and is only applicable to the specified I/O path. If you do not have an ON END statement in a program, the EOF/EOR condition will produce an error that is trappable by the ON ERROR statement. Encountering a logical or physical end of file will produce Error 59. Encountering an end of record in random mode produces Error 60.

You can have any number of ON END statements in a program context. ON END statements that refer to different I/O paths will not interfere with each other, even if the paths go to the same file. If you have more than one ON END to the same I/O path, the system will use whichever one it most recently executes during program flow.

An ON END is cancelled by the OFF END statement. OFF END only cancels the ON END branch for the specified I/O path. Re-ASSIGNing an I/O path will also cancel any existing ON END branch for the particular path.

The example below illustrates some of the more common situations that cause an EOF condition.

```
100   CREATE BDAT "ONEND",10,8
110   ASSIGN @Endpath TO "ONEND"
120   ON END @Endpath GOTO Eof1
130   FOR Inc=1 TO 20
140     OUTPUT @Endpath,Inc;SQR(Inc)
150   NEXT Inc
160 Eof1:  !
170   PRINT "EOF CONDITION -- ATTEMPT TO RANDOMLY WRITE BEYOND PHYSICAL END OF F
ILE."
180   PRINT
190   !
200   ON END @Endpath GOTO Eof2
210   OUTPUT @Endpath,5;"THIS IS A STRING."
220 Eof2:  !
230   PRINT "EOR CONDITION -- ATTEMPT TO RANDOMLY WRITE DATA   ITEM LONGER THAN
RECORD."
240   PRINT
250   !
260   ON END @Endpath GOTO Eof3
270   ENTER @Endpath,5;Str$
280 Eof3:  !
290   PRINT "EOR CONDITION -- ATTEMPT TO READ DATA ITEM LONGER THAN RECORD."
300   PRINT
310   !
```

```
320   ASSIGN @Endpath TO "ONEND"
330   ON END @Endpath GOTO Eof4
340   FOR Inc=1 TO 100
350     OUTPUT @Endpath;"A"
360   NEXT Inc
370 Eof4:  !
380   PRINT "EOF CONDITION -- ATTEMPT TO SERIALLY WRITE BEYOND PHYSICAL END OF F
ILE."
390   PRINT
400   !
410   ASSIGN @Endpath TO "ONEND"
420   ON END @Endpath GOTO Eof5
430   FOR Inc=1 TO 100
440     ENTER @Endpath;Str$
450   NEXT Inc
460 Eof5:  !
470   PRINT "EOF CONDITION -- ATTEMPT TO SERIALLY READ BEYOND  PHYSICAL END OF F
ILE."
480   PRINT
490   !
500   ON END @Endpath GOTO Eof6
510   OUTPUT @Endpath,5;5,END
520   ENTER @Endpath,6;X
530 Eof6:  !
540   PRINT "EOF CONDITION -- ATTEMPT TO RANDOMLY READ BEYOND  LOGICAL END OF FI
LE."
550   !
560   END
```

```
EOF CONDITION -- ATTEMPT TO RANDOMLY WRITE BEYOND
PHYSICAL END OF FILE.

EOR CONDITION -- ATTEMPT TO RANDOMLY WRITE DATA
ITEM LONGER THAN RECORD.

EOR CONDITION -- ATTEMPT TO READ DATA ITEM LONGER
THAN RECORD.

EOF CONDITION -- ATTEMPT TO SERIALLY WRITE BEYOND
PHYSICAL END OF FILE.

EOF CONDITION -- ATTEMPT TO SERIALLY READ BEYOND
PHYSICAL END OF FILE.

EOR CONDITION -- ATTEMPT TO RANDOMLY READ BEYOND
LOGICAL END OF FILE.
```

This example highlights a number of interesting points. First, in line 210 we try to randomly write a 17-byte string into an 8-byte record. The system returns an EOR condition. The length header for the string, however, is still 17. So when we try to read the string in line 270, we again receive an EOR condition.

In line 320 we re-ASSIGN the I/O path name in order to position the file pointer to byte 1. Then we redefine the ON END branch. These two statements must appear in this order since re-ASSIGNing an I/O path has the effect of canceling any ON END branch previously associated with the path.

In line 510, we shrink the file by moving the EOF pointer to the end of record 5 with the "END" secondary word. When we try to read record 6 in line 520 we get an EOF condition.

Protecting Files

Protect codes are two-character strings that can be assigned to any BDAT, BIN or PROG type file with the PROTECT statement. Protect codes are not unbreakable; they are only intended to prevent accidentally writing in files and directories.

For instance, the following statement assigns the protect code "AA" to the file named "FILE1."

```
PROTECT "FILE1","AA"
```

File specifiers in mass storage statements that write to a file or directory must include the protect code, if the file has one. Mass storage statements that read a file or directory do not require the protect code (e.g., CAT, LOAD, LOAD BIN, LOADSUB ALL FROM, GET and COPY). A protect code is specified by placing it in brackets right after the file name. To assign an I/O path name to the file named "FILE1," you would now have to include the protect code.

```
ASSIGN @Path1 TO "FILE1<AA>"
```

If you assign a protect code longer than two characters, the system will ignore everything after the second (non-blank) character. For example, the protect codes LONGPASS, LOLLYPOP, and LOST all result in the same protect code: LO. This rule holds both for PROTECTing a file and for specifying the protect code in a file specifier. For instance:

```
PROTECT "FILE1","Protect1"
```

would assign the protect code "Pr" to FILE1. To rename the file, we could write:

```
RENAME "FILE1<Prattle>" TO "FILE2"
```

"Prattle" is an acceptable protect code, since it starts with "Pr." Note that we do not include a protect code in the new file name. If you do, the system ignores it since the old protect code is passed to the new file name. FILE2 still has the protect code "Pr". To rename the file again, we might write:

```
RENAME "FILE2<Pr>" TO "FILE3"
```

Renaming a file has the effect of changing the file name in the directory and leaving everything else intact.

In addition to using the PROTECT statement, you can also assign a protect code to a BDAT file when you create it. For example:

```
CREATE BDAT "Example<xx>",10
```

creates a 10-record BDAT file called "Example" and gives it a protect code of "xx". You can also do this to PROG files with the STORE and STORE BIN statements. However, since ASCII files cannot be protected, a protect code cannot be included in any CREATE ASCII, SAVE, or RE-SAVE statement.

To change a protect code, simply execute a new PROTECT statement. To change the protect code of "Example" to "yy," execute:

```
PROTECT "Example<xx>","yy"
```

Note that you must include the current protect code in the file specifier.

To completely remove a protect code from a file, PROTECT the file with a code consisting of two blanks. For example, to remove the protect code from file "Example," execute:

```
PROTECT "Example<yy>","  "
```

When specifying a file that does not have a protect code, you can either ignore the code entirely or include a code of two spaces:

```
PURGE "Example"
                or
PURGE "Example<  >"
```

Securing Program Lines

While PROTECT prevents unintentional writing into files and directories, it does not prevent a programmer from looking at lines in a program. The SECURE statement prevents program lines from being listed.

For example, the following statement secures lines 10 thru 100 of the program currently in memory.

```
SECURE 10,100
```

If you want the entire program to be secure just execute:

```
SECURE
```

Note

Once a program is secured, it cannot be UNsecured. You should keep a backup copy of all programs in their unsecured form.

When you list a program, the secured lines are listed with asterisks after the line numbers. For example, lines 30 thru 60 are secured in the following listing.

```
10   !Example of Secured
20   !Begin password check
30*
40*
50*
60*
70   !End of password check
80
```

Copying Files and Volumes

The COPY statement allows you to duplicate individual files or an entire disc volume. Any type of file may be copied.

COPY of a file duplicates the existing file and places the new file name in the directory. A new file can be created either on the same disc or on another disc. If you copy a file to the same disc, the new file name must be different from the existing file name. If the file is of BDAT, BIN or PROG type, you can also assign a protect code to the new file. If there is not enough room on the disc for the file to be copied, the system cancels the statement and returns an error.

If the COPY statement specifies two mass storage volumes and no file names, a copy of the entire source volume is created. The purpose of this COPY option is to duplicate discs or make back-up copies. Note that you will **lose any old information** in the directory of the destination volume. A volume copy does not append to the contents of the destination; when the copy is finished, the source and destination volumes will be identical. You cannot copy a larger volume to a smaller volume, regardless of the amount of data on the larger volume. You can quickly purge all the files on a volume by copying an empty disc onto a full disc.

Examples

The following statement copies "File1" from the current system mass storage device to a new file called "File2" on the same mass storage.

```
COPY "File1" TO "File2"
```

The following statement copies "File1" from the current system mass storage to the drive at interface select code 7, primary address 0, unit number 1. Note that both files can be named "FILE1" if they are on different volumes.

```
COPY "File1" TO "File1:HP,700,1"
```

The following statement copies a file from an HP 82901 drive to the current system mass storage device. The new file "DATA" is given the protect code "xx."

```
COPY "File1:HP82901,700,0" TO "DATA<xx>"
```

The following statement copies the entire disc from the right-hand internal drive to the left-hand drive of a Model 236.

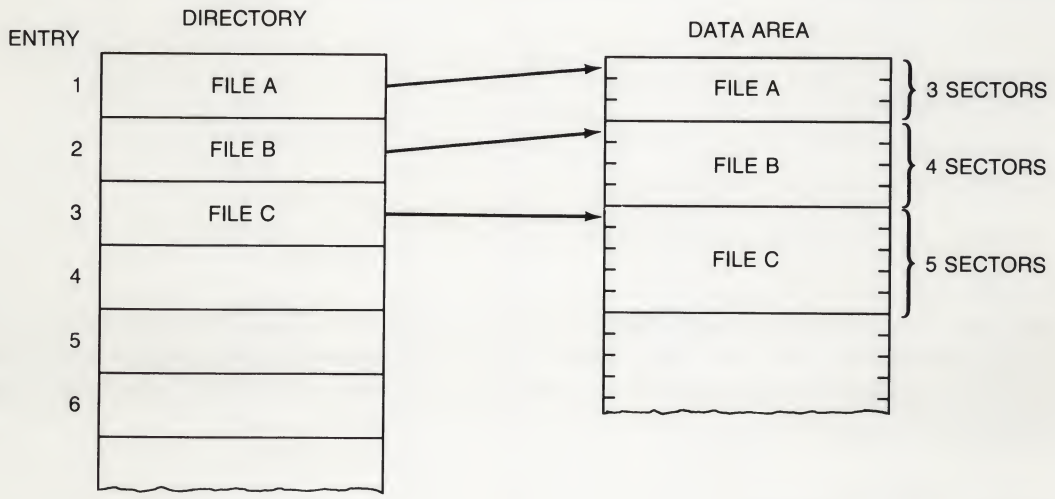
```
COPY ":INTERNAL" TO ":INTERNAL,4,1"
```

Purging Files

You can purge a file from the directory by using the PURGE statement. Purging a file deletes the directory entry for the file and releases the reserved space in the data area. Purging a file, therefore, creates two "gaps" on the disc: one in the data area and one in the directory. When you create a file, the system looks at all the gaps in the data area to see if the newly created file will fit in any of them.

Directory entries must be in the same order as the files in the data area. The fourth directory entry, for example, must correspond to the fourth file in the data area. Consequently, if you PURGE a file, and then create a smaller file, you may lose disc space. The following examples illustrate this principle.

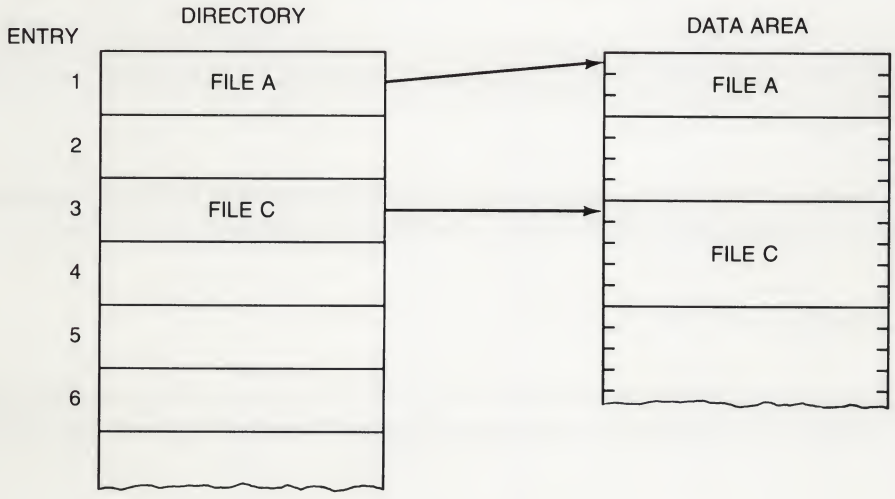
Suppose that you have three consecutive files on a disc with the following names and sizes.



Executing the following statement:

```
PURGE "FILEB"
```

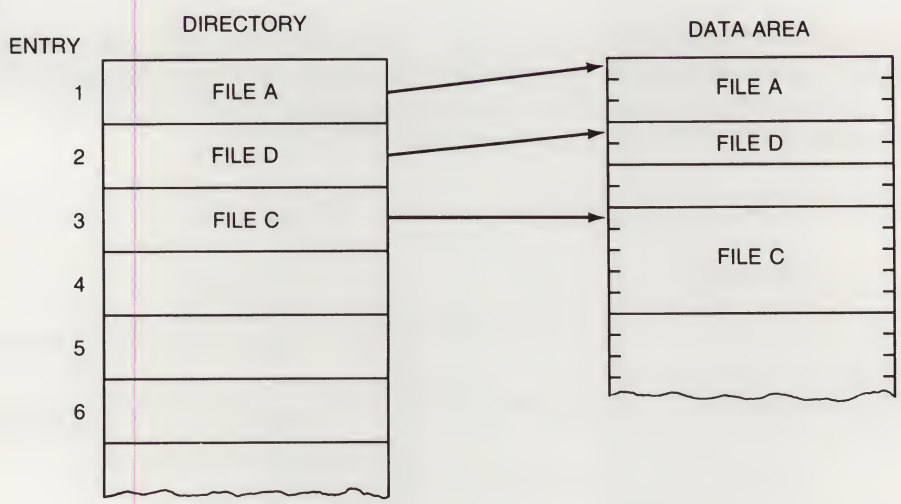
creates a 1-entry gap in the directory and a 4-sector gap in the data area.



Now, suppose you create a 2-sector file:

```
CREATE ASCII "FILED",2
```

The system will place this file in the data-area gap and place the directory entry in the directory gap.



You now have a 2-sector gap in the data area but no gaps in the directory. If you create another file, the system will fill entry 4 in the directory and will reserve space in the data area past FILEC. The two unused sectors will not be reclaimed unless you PURGE one of the adjacent files, FILED or FILEC.

Accessing Directories

Disc structure and mass storage directories were briefly described earlier in this chapter. As you may recall, a directory is merely an index to the files on a mass storage media. The BASIC language has several features that allow you to obtain information from the directories of mass storage media. This section presents several techniques that will help you access this information.

To get a catalog listing of a directory, you will use the CAT statement. Executing CAT with no media specifier directs the system to get a catalog of the current system mass storage directory.

```
CAT
```

Including a media specifier directs the system to get a catalog of the specified mass storage. For instance, executing the following statement returns a catalog of the directory of the left drive of a Model 236.

```
CAT ":INTERNAL,4,1"
```

Both of the preceding statements sent the catalog listings to the current system printer (the one specified in the last PRINTER IS statement; the default system printing device is the CRT).

Sending Catalogs to External Printers

The CAT statement normally directs its output to the current PRINTER IS device. The CAT statement can also direct the catalog to a specified device, as shown in the following examples:

```
CAT TO #701
CAT TO #External_prtr
CAT TO #Device_selector
```

The parameter following the # is known as a device selector and is further described in Chapter 8, "Using a Printer," and in the Glossary of the *BASIC Language Reference.*

Extended Access of Directories

With MS, the CAT statement has the following additional capabilities:

- additional information about PROG files may be obtained
- the mass storage directory can be sent to a string array
- files to be cataloged may be selected by name or by beginning letter(s) of the file name
- the number of selected file entries may be counted
- the CAT operation may be directed to "skip" a specific number file entries before sending entries to the destination
- the catalog header may be suppressed

Cataloging Individual PROG Files

Performing CAT operations on an individual PROG file returns additional information about the file. A catalog of a PROG files yields the following information:

- a list of the binary program(s) contained in the program file and the size of each (in bytes)
- the size of the main program (in bytes).
- a list of contexts (SUB and FN subprograms) and their sizes (in bytes)

The following catalog listing is an example of a CAT performed on an individual PROG file. Note that this catalog format only requires 45 columns.

```
 NEWPAGER_A
NAME                  SIZE TYPE
===================== ===== ================
MAIN                 62002 BASIC
FNBar$                3680 BASIC
FNRoman$               656 BASIC
Killkeys               426 BASIC
FNTrim$                414 BASIC
FNUpc$                 344 BASIC
FNLwc$                 416 BASIC
Table_formatter       6810 BASIC
Strip                 1260 BASIC
AVAILABLE ENTRIES = 0
```

The `AVAILABLE ENTRIES` table entry is not currently used.

If the program contains a binary or PHYREC program, a warning and the version codes of both the BASIC system and the binary program are included in the catalog information. PHYREC programs are not supported with BASIC 3.0. The following example shows the format of the message returned.

```
Prog_phy
NAME                      SIZE TYPE
========================= ===== ================
PHYREC 2.0                1734 BASIC BINARY
*** WARNING:  System level 3.  Bin level 2.
MAIN                       222 BASIC
AVAILABLE ENTRIES = 0
```

Cataloging to a String Array

The following example program segment shows an example of directing the catalog of mass storage file entries to the CRT and then to a string array.

```
100   PRINT "          CAT to CRT."
110   PRINT "-----------------------------------"
120   CAT TO #CRT;COUNT Files_and_headr  ! Includes 5-line header.
130   PRINT "Number of files=";Files_and_headr-5
140   PRINT
150   !
160   PRINT "     CAT to a string array."
170   PRINT "-----------------------------------"
180   Array_size=Files_and_headr+2  ! Allow for 7-line header.
190   ALLOCATE Catalog$(1:Array_size)[80]
200   CAT TO Catalog$(*)
210   FOR Entry=1 TO Array_size
220     PRINT Catalog$(Entry)
230   NEXT Entry
240   PRINT "Number of files=";Array_size-7
250   PRINT
260   !
270   END
```

The program produces the following output.

```
 CAT to CRT.
-----------------------------------
:INTERNAL
VOLUME LABEL: B9826
FILE NAME PRO TYPE  REC/FILE BYTE/REC   ADDRESS
Data1         ASCII         3      256        16
Char1         BDAT          3      256        20
Prog1         PROG          2      256        23
Char2         BDAT          7      256        26
Prog2         PROG          2      256        33
Data2         ASCII         9      256        35
Char3         BDAT          6      256        45
Data3         ASCII         5      256        51
BCD_INTR      ASCII         3      256        56
BCD_CONFIG    ASCII         9      256        59
BCD_ENT1      ASCII         2      256        68
BCD_OUT1      ASCII         1      256        70
BCD_ENTBIN    ASCII         2      256        71
BCD_ENTFMT    ASCII        10      256        73
Number of files= 14

CAT to a string array.
-----------------------------------
:INTERNAL, 4
LABEL:  B9826
FORMAT: LIF
AVAILABLE SPACE:   892
SYS  FILE   NUMBER   RECORD     MODIFIED     PUB OPEN
FILE NAME             LEV TYPE  TYPE  RECORDS   LENGTH DATE        TIME ACC STAT
==================== === ==== ===== ======== ======== ================ === ====
Data1                  1       ASCII         3      256                  MRW
Char1                  1 98X6 BDAT          3      256                  MRW
Prog1                  1 98X6 PROG          2      256                  MRW
Char2                  1 98X6 BDAT          7      256                  MRW
Prog2                  1 98X6 PROG          2      256                  MRW
Data2                  1       ASCII         9      256                  MRW
Char3                  1 98X6 BDAT          6      256                  MRW
Data3                  1       ASCII         5      256                  MRW
BCD_INTR               1       ASCII         3      256                  MRW
BCD_CONFIG             1       ASCII         9      256                  MRW
BCD_ENT1               1       ASCII         2      256                  MRW
BCD_OUT1               1       ASCII         1      256                  MRW
BCD_ENTBIN             1       ASCII         2      256                  MRW
BCD_ENTFMT             1       ASCII        10      256                  MRW
Number of files= 14
```

Including the keyword COUNT followed by a numeric variable returns the total number of file entries plus header lines to that variable; here, the variable `Files_and_headr` is used. In this example, a value of 2 is added to this variable to compensate for the 7-line header which is sent instead of the usual 5-line header. This new value, stored in `Array_size`, is then used to direct the computer to ALLOCATE just enough space in a string-array variable to hold the directory listing. The program can then search the directory listing for further information, if desired.

You may have noticed that the format for catalogs sent to string arrays (the second catalog listing) is different from catalogs sent to the PRINTER IS device. This catalog format requires that each array element must be dimensioned to hold at least 80 characters with this type of CAT operation. Again, the header contains 7 lines, not 5 as with catalogs sent to devices.

If the CAT operation would not have filled the string array, the unused array elements would have been set to the null string (i.e., strings of length 0). If there are more catalog lines than string-array elements, the operation stops when the array is filled. No indication of the "overflow" is reported; the count returned is equal to the number of array elements.

Suppressing the Catalog Header

To suppress the catalog header that would otherwise be sent automatically to the destination, use the following syntax:

```
CAT;NO HEADER
CAT TO String_array$(*);NO HEADER
CAT "Prog_2";NO HEADER
```

Using NO HEADER suppresses the 5-line or 7-line heading of each catalog format shown above, respectively. The catalog listing of a PROG file would be 3 lines shorter. The first line of each catalog listing contains the first directory entry, the second element contains the second entry, and so forth.

Cataloging Selected Files

The directory entry of file(s) that begin with certain character(s) can be obtained by using the secondary keyword SELECT. For this example, assume that the directory contains the following entries:

```
 :INTERNAL
VOLUME LABEL: B9826
FILE NAME PRO TYPE  REC/FILE BYTE/REC   ADDRESS
Data1         ASCII        3      256        16
Char1         BDAT         3      256        20
Prog1         PROG         2      256        23
Char2         BDAT         7      256        26
Prog2         PROG         2      256        33
Data2         ASCII        9      256        35
Char3         BDAT         6      256        45
Data3         ASCII        5      256        51
BCD_INTR      ASCII        3      256        56
BCD_CONFIG    ASCII        9      256        59
BCD_ENT1      ASCII        2      256        68
BCD_OUT1      ASCII        1      256        70
BCD_ENTBIN    ASCII        2      256        71
BCD_ENTFMT    ASCII       10      256        73
```

Suppose that you want to catalog only files beginning with the letters "Prog". The following examples show how this may be accomplished. Notice that this is **not** the same operation as getting a catalog of a PROG file.

```
Beginning_chars$="Prog"
CAT;SELECT Beginning_chars$

CAT;SELECT "Prog",COUNT Files_and_headr
```

The directory entries of the three files beginning with the letters "Prog" are sent to the PRINTER IS device. In the second CAT statement above, the variable `Files_and_headr` is filled with the number of selected files found on the current default mass storage device (plus the 5 header lines). (Keep in mind that the variable `Files_and_headr` must be currently defined in the current program context.)

The following result would be sent to the system printing device.

```
:INTERNAL, 4
VOLUME LABEL: B9826
FILE NAME PRO TYPE  REC/FILE BYTE/REC   ADDRESS
Prog1         PROG         2      256        23
Prog2         PROG         2      256        33
Prog3         PROG         2      256       533
```

SELECT may also be used to get the catalog of an individual file entry by selecting the entire file name, as shown in the following statement:

```
CAT;SELECT "Char3"
```

Getting a Count of Selected Files

It is often desirable to determine the total number of files on a disc, or the number that begin with a certain character or group of characters. The COUNT option directs the computer to return the number of selected files in the variable that follows the COUNT keyword.

```
CAT;COUNT Files_and_headr
CAT;SELECT "Data",COUNT Selected_files
```

The first CAT operation returns a count of all files in the directory (plus 5 header lines), since not including SELECT defaults to "select all files". The second operation returns a count of the specifically selected files (plus 5).

Skipping Selected Files

If there are many files that begin with the same characters, it is often useful to be able to skip some of the directory entries so that the catalog is not as long. This may be especially useful when using a drive such as an HP 7912, which has the capability of storing more than 10 000 files.

The following statement shows an example of skipping file entries before sending selected entries to the destination.

```
CAT;SELECT "BCD",SKIP 5

:INTERNAL, 4
VOLUME LABEL: B9826
FILE NAME PRO TYPE  REC/FILE BYTE/REC   ADDRESS
BCD_ENTFMT    ASCII        10      256        73
```

The first five "selected" files (that begin with the specified characters) are "skipped" (i.e., not sent with the rest of the catalog information).

Including COUNT in the previous CAT operation (as shown below) returns a count of the selected files (plus header lines), not just the catalog lines sent to the destination. Remember that selected files includes all files skipped, if any. In this case, a value of 11 is returned, not 1 (or 6) as might be expected.

```
CAT;SELECT "BCD",SKIP 5,COUNT Catalog_lines
```

Note that if SKIP is included, the count remains the same (as long as at least one file is cataloged). If the number of files to be skipped equals the number of files selected, COUNT returns a value of zero.

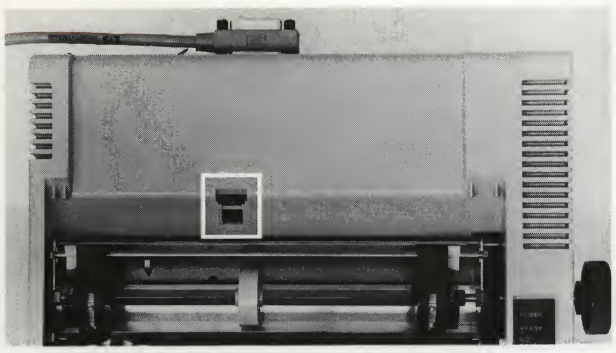
```
CAT;SELECT "BCD",SKIP 6,COUNT Files_and_headr
```

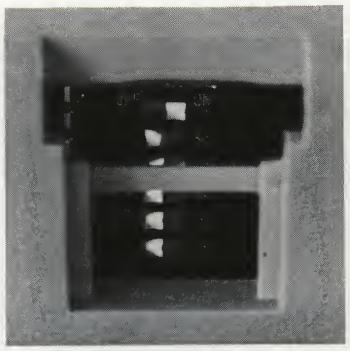
The following program shows an example of looking at the files in a catalog by viewing a small "window" of files at one time. The technique is useful for decreasing the amount of memory required to hold a catalog listing in a string array.

```
100   ! Declare a small string array (7 elements).
110   DIM Array$(1:7)[80]
120   !
130   ! Send header to the array.
140   CAT TO Array$(*)
150   ! Print header.
160   FOR Element=1 TO 7
170     PRINT Array$(Element)[1,45]
180   NEXT Element
190   !
200   ! Now get 7-line "windows" and print files therein.
210   First_file=1  ! Begin with first file in directory.
220   REPEAT  ! Send file entries to Array$ until last file sent.
230     !
240     ! Send files to Array$; SKIP files already printed;
250     ! return index (with COUNT) of last file sent to Array$.
260     CAT TO Array$(*);SKIP First_file-1,COUNT Last_file,NO HEADER
270     DISP "First file=";First_file;"; Last file=";Last_file
280     !
290     ! Print file entries (no entry printed when Last_file=0).
300     FOR Element=1 TO (Last_file-First_file)+1  ! (6 or less)+1.
310       PRINT Array$(Element)[1,45]
320     NEXT Element
330     !
340     First_file=Last_file+1  ! Point to next "window."
350     !
360   UNTIL Last_file=0  ! Until SKIP >= number of files.
370   !
380   END
```

It is also important to note the order of options in the CAT statement. This order is required when several options are used. If the NO HEADER option is used, it must be the last option in the list, as shown in the following example.

```
CAT;SELECT "BCD",SKIP 5,COUNT Selected_files,NO HEADER
```





The primary address, determined by the switch settings, is combined with the interface select code to make up the device selector. In the following example the primary address **01** is appended to the interface select code **7** to produce the device selector **701**.

```
PRINTER IS 701
```

This statement tells the computer to use a the internal HP-IB interface (select code 7) to communicate with a printer whose switches are set to the primary address "01". If the printer's primary address is set to "11", the device selector would be "711".

A device selector can be created mathematically by multiplying the interface select code by 100 and adding the primary address. For example, a printer on the internal HP-IB bus whose primary address is set to 9 would have the device selector 709 ($7 \times 100 + 9 = 709$).

Switch Settings

Five switch segments, dedicated to setting the primary address, allow thirty-two possible addresses. In the following decimal-to-binary conversion table, each binary digit corresponds to one of the switch segments. A '1' indicates the switch segment is **on**, while a '0' indicates the switch segment is **off**.

Decimal	Binary	Decimal	Binary
0	00000	16	10000
1	00001	17	10001
2	00010	18	10010
3	00011	19	10011
4	00100	20	10100
5	00101	21	10101
6	00110	22	10110
7	00111	23	10111
8	01000	24	11000
9	01001	25	11001
10	01010	26	11010
11	01011	27	11011
12	01100	28	11100
13	01101	29	11101
14	01110	30	11110
15	01111	31	11111

A quick glance at the switch segments lets you confirm the primary address.

Using Device Selectors

A device selector is used by several different statements. In each of the following, the numeric constant is a device selector.

`PRINTER IS 1`	Specifies the internal CRT. (default)
`PRINTER IS 701`	Specifies a printer with interface select code 7 and switch selected to primary address 01.
`PRINTER IS 22`	Specifies a printer connected through interface select code 22.
`CAT TO #701`	Prints a disc directory at 701.
`PRINTALL IS 707`	Logs information on a printer whose select code is 7 and whose switches are set to primary address 07 (binary 00111).
`LIST #701`	Lists the program in memory to a HP-IB printer set to primary address 01.

Most statements allow a device selector to be assigned to a variable. Either INTEGER or REAL variables may be used.

```
PRINTER IS Hal
CAT TO #Dog
```

The following three-letter mnemonic functions have been assigned useful values.

Mnemonic	Value
PRT	701
KBD	2
CRT	1

For example, the following statements perform the same action.

```
PRINTER IS PRT
PRINTER IS 701
```

The mnemonic may be used anywhere the numeric device selector can be used.

Another method may be used to identify the printer within a program. An I/O path name may be assigned to the printer; the printer is subsequently referenced by the I/O path name. This technique is fully explained in the *BASIC Interfacing Techniques* manual.

Chapter 8 Using a Printer

Introduction

Sooner or later it needs to be printed. A wide range of printers, supported by BASIC, can be connected to the Series 200 computers. This chapter covers the statements commonly used to communicate with external printers. The following list names some of the printers that work with Series 200 computers:

- HP 2631 Dot Matrix Printer
- HP 2671 Thermal Printer
- HP 2673 Intelligent Thermal Printer
- HP 9866 Thermal Printer (with interface)
- HP 9876 Thermal Graphics Printer
- HP 82905 Dot Matrix Printer
- HP 2601 Daisy-Wheel Printer

All these printers, except the HP 9866, can be directly connected to the computer's internal HP-IB interface. The HP 9866 requires its own interface.

Fundamentals

The PRINT statement normally directs text to the screen of the CRT. Text may be re-directed to an external printer by using the PRINTER IS statement. The default **system printer** is the screen of the CRT. The PRINTER IS statement is used to change the system printer.

Before a printer will print the first character, several steps are required to set up the printer. These steps are fully documented in the appropiate printer installation manual.

After the printer is switched on and the computer and printer have been connected via an interface cable, there is only one piece of information needed before printing can begin. The computer needs to know the correct **device selector** for the printer. This is analogous to knowing the correct telephone number before making a call.

Device Selectors

A device selector is a number that uniquely identifies a particular device connected to the computer. When only one device is allowed on a given interface, it is uniquely identified by the **interface select code**. In this case, the device selector is the same as the interface select code.

For example, the internal CRT is the only device at the interface whose select code is 1. To direct the output of PRINT statements to the CRT, use the following statement.

```
PRINTER IS 1
```

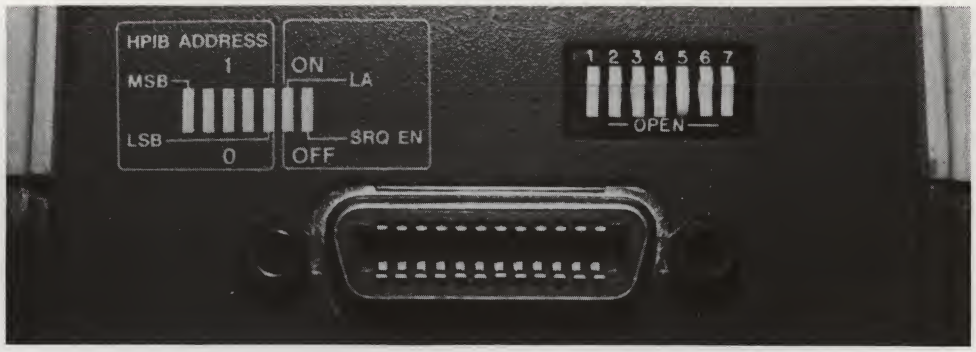
This statement defines the screen of the CRT to be the system printer. Until changed, the output of PRINT statements will appear on the screen of the CRT.

When more than one device can be connected to an interface, such as the internal HP-IB interface, (interface select code 7) the interface select code no longer uniquely identifies the printer. Extra information is required. This extra information is the **primary address**.

Primary Addresses

Each printer has a set of switches, usually located on the back panel, which determine the primary address of the printer.

The following photographs show the switch locations on various printers. In addition to the primary address switch segments there are usually segments that control the printers response to other signals on the HP-IB bus.





Using the External Printer

Most ASCII characters are printed on an external printer just as they appear on the screen of the CRT. Depending on your printer, there will be exceptions. Several printers will also support an alternate character set: either a foriegn character set, a graphics character set, or an enhanced character set. If your printer supports an alternate character set, it usually is accessed by sending a special command to the printer.

Control Characters

In addition to a "printable" character set, printers usually respond to control characters. These non-printing characters generally produce a response from the printer. The following table shows some of the control characters and their effect.

Printer's Response	Control Character	ASCII Value
ring printer's bell	ctrl-G	7
backspace one character	ctrl-H	8
horizontal tab	ctrl-I	9
line-feed	ctrl-J	10
form-feed	ctrl-L	12
carriage-return	ctrl-M	13

One way to send control characters to the printer is the CHR$ function. Execute the following.

```
PRINTER IS 701
PRINT CHR$(12)
```

The printer responds with a formfeed. To resume printing on the internal CRT, execute the following.

```
PRINTER IS 1
PRINT "Back to the CRT."
```

Other control characters may be valid for your printer. For example, sending a control-N to the HP 82905A printer changes the character size (font) of subsequent text.

```
10 PRINTER IS 701
20 Big$=CHR$(14)
30 PRINT BIT$;"Double-Width Text"
40 PRINTER IS CRT
50 END
```

Refer to the appropriate printer manual for a complete listing of control characters and their effect on your printer. Some control characters will only affect the current line of text.

Escape-Code Sequences

Similar in use to control characters, escape-code sequences allow additional control over most printers. These sequences consist of the escape character, CHR$(27), followed by one or more characters.

For example, the HP 2631 printer is capable of printing characters in several different fonts. To print extended characters on the HP 2631 an escape code sequence is sent to the printer before the actual text to be printed is sent.

```
10  PRINTER IS 701
20  Esc$=CHR$(27)
30  Big$="&k1S"
40  Regular$="&k0S"
50  PRINT Esc$;Big$;"Extended-Font Text"
60  PRINT Esc$;Regular$;"Back to normal."
70  PRINTER IS 1
80  END
```

Many escape code sequences can be used by more than one printer. For instance, the HP 2671 and the HP 2631 share the same escape code sequence for underlining text.

```
10    PRINTER IS PRT
20    Under$=CHR$(27)&"&dD"
30    Normal$=CHR$(27)&"&d@"
40    PRINT "This is not underlined"
50    PRINT Under$&"This is underlined"&Normal$
60    PRINT "Done."
70    PRINTER IS CRT
80    END
```

Since each printer may respond differently to control characters and escape code sequences, check the manual that came with your printer for details concerning their use.

Formatted Printing

For many applications the PRINT statement provides adequate formatting. The simplest method of print formatting is by specifying a comma or semicolon between printed items.

When the comma is used to seperate items the printer will print the items on field boundries. Fields start in column one and occur every ten columns (columns 1,11,21,31,...). Using the values: A=1.1 B=−22.2 C=3E+5 D=4.4E+8

```
PRINT A,B,C,D
```

Produces:

```
12345678901234567890123456789012345678901234567890123456789
 1.1         -22.2         300000      5.1E+8
```

Note the form of numbers in a normal PRINT statement. A positive number has a leading and a trailing space printed with the number. A negative number uses the leading space position for the "−" sign. This is why the positive numbers in the previous example appear to print one column to the right of the field boundries. The next example shows how this form prevents numeric values from running together.

```
PRINT A;B;C;D,E
12345678901234567890123456789012345678901234567890123
 1.1 -22.2  300000  5.1E+8
```

Using the semicolon as the separater caused the numbers to be printed as closely together as the "compact" form allows. The compact form always uses one leading space (except when the number is negative) and one trailing space.

The comma and semicolon are often all that is needed to print a simple table. By using the ability of the PRINT statement to print the entire contents of of a array, the comma or semicolon can be used to format the output.

If each array element contained the value of its subscript, the statement:

```
PRINT Array(*);
```

Produces:

```
 0  1  2  3  4  5  6  7  8  9  10  11  12  13  14 ...
```

Another method of aligning items is to use the tabbing ability of the PRINT statement.

```
PRINT TAB(25);-1.414
12345678901234567890123456789012345678901234567890123
                         -1.414
```

While PRINT TAB works with an external printer, PRINT TABXY will not. PRINT TABXY may be used to specify both the horizontal and vertical position when printing to the internal CRT.

A more powerful formatting technique employs the ability of the PRINT statement to allow an image to specify the format.

Using Images

Just as a mold is used for a casting, an image can be used to format printing. An image specifies how the printed item should appear. The computer then attempts to print the item according to the image.

One way to specify an image is to include it in the PRINT statement. The **image specifier** is enclosed within quotes and consists of one or more **field specifiers**. A semicolon then separates the image from the items to be printed.

```
PRINT USING "D.DDD";PI
```

This statement prints the value of pi (3.141592659...) rounded to three digits to the right of the decimal point.

```
3.142
```

For each character "D" within the image, one digit is to be printed. Whenever the number contains more non-zero digits to the right of the decimal than provided by the field specifier, the last digit is rounded. If more precision is desired, more characters can be used within the image.

```
PRINT USING "D.10D";PI
3.1415926536
```

Instead of typing ten "D" specifiers, one for each digit, a shorter notation is to specify a repeat factor before each field specifier character. The image "DDDDDD" is the same as the image "6D".

The image specifier can be included in the PRINT statement or on it's own line. When the specifier is on a different line, the PRINT statement accesses the image by either the line number or the line label.

```
100    Format: IMAGE "6Z.DD"
110    PRINT USING Format;A,B,C
120    PRINT USING 100;A,B,C
```

Both PRINT statements use the image in line 100.

Numeric Image Specifiers

Several characters may be used within an image to specify the appearance of the printed value.

Image Specifier	Purpose
D	Replace this specifier with one digit of the number to be printed. If the digit is a leading zero, print a space. if the value is negative, the position may be used by the negative sign.
Z	Same as "D" except that leading zeros are printed.
E	Prints two digit of the exponent after printing the sequence "E+". This specifier is equal to "ESZZ". See the Language Reference for more details.
K	Print the entire number without leading or trailing spaces.
S	Print the sign of the number: either a "+" or "−".
M	Print the sign if the number is negative; if positive, print a space.
.	Print the decimal point.
H	Similar to K, except the number is printed using the European number format (comma radix). (Requires IO)
R	Print the comma (European radix) (Requires IO)
*	Like Z, except that asterisks are printed instead of leading zeros. (Requires IO)

To better understand the operation of the image specifiers examine the following examples and results.

Statement	Output
`PRINT USING "K";33.666`	`33.666`
`PRINT USING "DD.DDD";33.666`	`33.666`
`PRINT USING "DDD.DD";33.666`	`33.67`
`PRINT USING "ZZZ.DD";33.666`	`033.67`
`PRINT USING "ZZZ";.444`	`000`
`PRINT USING "ZZZ";.555`	`001`
`PRINT USING "SD.3DE";6.023E+23`	`+6.023E+23`
`PRINT USING "S3D.3DE";6.023E+23`	`+602.300E+21`
`PRINT USING "S5D.3de";6.023E+23`	`+60230.000E+19`
`PRINT USING "H";3121.55`	`3121,55`
`PRINT USING "DDRDD";19.95`	`19,95`
`PRINT USING "***";.555`	`**1`

To specify multiple fields within the image, the field specifiers are separated by commas.

Statement	Output
`PRINT USING "K,5D,5D";100,200,300`	`100  200  300`
`PRINT USING "DD,ZZ,DD";1,2,3`	`102 3`

If the items to be printed can use the same image, the image need be listed only once. The image will then be re-used for the subsequent items.

```
PRINT USING "5D.DD";3.98,5.95,27.50,139.95


12345678901234567890123456789012345
    3.98    5.95   27.50  139.95
```

The image is re-used for each value. An error will result if the number cannot be accurately printed by the field specifier.

String Image Specifiers

Similar to the numeric field image characters, several characters are provided for the formatting of strings.

Image Specifier	Purpose
A	Print one character of the string. If all characters of the string have been printed, print a trailing blank.
K	Print the entire string without leading or trailing blanks
X	Print a space.
"literal"	Print the characters between the quotes.

The following examples show various ways to use string specifiers.

```
PRINT USING "5X,10A,2X,10A";"Tom","Smith"

12345678901234567890123456789012345
     Tom          Smith

PRINT USING "5X,""John"",2X,10A";"Smith"

12345678901234567890123456789012345
     John  Smith

PRINT USING """PART NUMBER"",2x,10d";90001234

12345678901234567890123456789012345
PART NUMBER    90001234
```

Additional Image Specifiers

The following image specifiers serve a special purpose.

Image Specifier	Purpose
B	Print the corresponding ASCII character. This is similar to the CHR$ function.
#	Suppress automatic end-of-line sequence.
L	Send the current end-of-line (EOL) sequence; with IO, see the PRINTER IS statement in the *BASIC Language Reference* manual for details on re-defining the EOL sequence.
/	Send a carriage-return and a linefeed.
@	Send a formfeed.
+	Send a carriage-return as the EOL sequence. (Requires IO)
–	Send a linefeed as the EOL sequence. (Requires IO)

For example:

`PRINT USING "@,#"` outputs a formfeed.

```
PRINT USING "D,X,3A,""OR NOT"",X,B,X,B,B";2,"BE",50,66,69
```

Special Considerations

If nothing prints, check if the printer is ON LINE. When the printer if OFF LINE the computer and printer can communicate but no printing will occur.

Sending text to a non-existent printer will cause the computer to wait indefinitely for the printer to respond. ON TIMEOUT may be used within a program to test for the printer. To clear the error press (CLR I/O), check the interface cable, and switch settings then try again.

Since the printer's device selector may change, keep track of the locations in the program where a device selector is used. If most of the program's output is sent to a printer, you may wish to use the PRINTER IS statement at the beginning of the program and then send messages to the CRT screen by using the OUTPUT statement.

```
PRINTER IS 701
PRINT "Text to the printer."
OUTPUT 1;"Screen Message"
PRINT "Back to the printer."
```

If the program must use the PRINTER IS statement frequently, assign the device selector to a variable; then if the device selector changes, only one program line will need to be changed.

The Real-Time Clock

Chapter 9

Introduction

The internal clock provides the date, time of day, and interval timing to one-hundredth of a second resolution. In this chapter you will find the statements and functions used to set and read the clock. Additional statements, allow you to define event-initiated branches based on the clock's value.

Many of the statements presented in this chapter require CLOCK.

Internally, the clock maintains the year, month, day, hour, minute, and second as a single real number. This number is scaled to an arbitrary "dawn of time" thus allowing it to also represent the Julian date. The current value of the clock is returned by the TIMEDATE function.

```
PRINT TIMEDATE
```

While the value returned contains all the information necessary to uniquely specify the date and time to the nearest one-hundredth of a second, it needs to be "unpacked" to provide understandable information.

The following functions are available to extract the date and time of day from TIMEDATE.

The DATE$ function extracts the date from the value of TIMEDATE.

```
PRINT DATE$(TIMEDATE)
```

Prints: `1 Mar 1900`

This is the default power-up date for machines without the powerfail option.

The TIME$ function returns the time of day.

```
PRINT TIME$(TIMEDATE)
```

Prints: `00:05:27`

Until the clock has been set to the correct time of day, the TIME$ function returns the time since power-up (on machines without the powerfail option).

The SET TIMEDATE statement is used to set the clock

```
SET TIMEDATE  DATE("1 OCT 1982") + TIME("8:37:30")
```

The time of day can be changed without affecting the date by the SET TIME statement.

```
SET TIME  TIME("9:55")
```

Clock Time

To minimize the space required to store the date and time, and yet insure a unique value for each point in time, both time and date are combined as a single real number. This value is the Julian date multiplied by the number of seconds in a day. By recalling that there are 86400 seconds in a day, the date and time of day can be extracted from TIMEDATE by the following simple alogrithms.

`TIMEDATE MOD 86400` returns the time of day, and

`TIMEDATE DIV 86400` returns the Julian date.

The time of day is expressed in seconds past midnight and is easily divided into hours, minutes, and seconds. The Julian date requires a bit more processing to extract the month, day, and year but this method insures a unique value for each day over the entire range of the clock (1 Mar 1900 through 4 Aug 2079).

Clock Time

Year	Clock Value
1900	2.086578144E+11
1910	2.089733472E+11
1920	2.092888800E+11
1930	2.096044992E+11
1940	2.099200320E+11
1950	2.102356512E+11
1960	2.105511840E+11
1970	2.108668032E+11
1980	2.111823360E+11
1990	2.114979552E+11
2000	2.118134880E+11
2010	2.121291072E+11
2020	2.124446400E+11
2030	2.127602592E+11
2040	2.130757920E+11
2050	2.133914112E+11
2060	2.137069440E+11
2070	2.140225632E+11
2080	2.143380960E+11

Hours	Seconds
1	3600
2	7200
3	10800
4	14400
5	18000
6	21600
7	25200
8	28800
9	32400
10	36000
11	39600
12	43200
13	46800
14	50400
15	54000
16	57600
17	61200
18	64800
19	68400
20	72000
21	75600
22	79200
23	82800
24	86400

Setting the Time

The time of day is changed by SET TIME X, where X is the number of seconds past midnight. The value of X must be in the range: 0 through 86399.99 seconds. The TIME function will convert twenty-four hour formatted time (HH:MM:SS) into the value needed to set the clock.

The TIME function converts an ASCII string representing a time of day, in twenty-four hour format, into the number of seconds past midnight. For example:

```
SET TIME TIME("15:30:10")
```

Is equivalent to:

```
SET TIME 55810
```

Either of these statements will set the time of day without changing the date. Use the SET TIMEDATE statement to change the date.

To display the new time, the TIME$ function formats the clock's value (TIMEDATE) into hours, minutes, and seconds.

```
PRINT TIME$(TIMEDATE)
```

Prints: `15:30:16`

Even though TIMEDATE returns a value containing both time of day and the Julian date, TIME$ performs an internal modulo 86400 on the value passed to the function and will always return a string in the range: `00:00:00` thru `23:59:59`.

The following program contains the routines to set and display the time of day. The routines are written as user-defined functions that may be appended to a program. Once appended to a program, the routines duplicate the TIME and TIME$ functions available with CLOCK. The formatted time can then be displayed by the following statement.

```
PRINT FNTime$(TIMEDATE)
```

Prints: `15:31:05`

Given the clock's value, the FNTime$ function returns the time of day in 24 hour format (HH:MM:SS). The FNTime function converts the time of day to seconds and is used to set the clock.

```
10 Show_time:DISP FNTime$(TIMEDATE)
20    GOTO Show_time
30    END
40    !
50    ! While the program is running, type:
60    ! SET TIME FNTIME("11:5:30")
70    ! then press <EXECUTE> to show the new time.
80    !
90    !*********************************************************
100   !
110   DEF FNTime$(Now) ! Given 'SECONDS' Return 'hh:mm:ss'
120     !
130     Now=INT(Now) MOD 86400
140     H=Now DIV 3600
150     M=Now MOD 3600 DIV 60
160     S=Now MOD 60
170     OUTPUT T$ USING "#,ZZ,K";H,":",M,":",S
180     RETURN T$
190   FNEND
200   !
210   DEF FNTime(T$) ! Given 'hh:mm:ss' Return 'SECONDS'
220     !
230     ON ERROR GOTO Err
240     ENTER T$;H,M,S
250     RETURN (3600*H+60*M+S) MOD 86400
260 Err:OFF ERROR
270     RETURN TIMEDATE MOD 86400
280   FNEND
```

After entering the program, follow the instructions at the beginning of the program to verify correct operation. Store this program in a file named "FUNTIME". The functions can be extracted from this program and appended to other programs by the LOADSUB statement.

Note that the FNTime function requires hours, minutes, and seconds, while the TIME function only requires hours and minutes.

Setting the Date

The date is changed by SET TIMEDATE X, where X is the Julian date multiplied by the number of seconds in a day (86400). The DATE function converts a formatted date (DD MMM YYYY) into the value needed to set the clock. Due to the wide range of values allowed by the DATE function, negative years can be specified, but not when using the function to set the clock.

The following statement will set the clock to the proper date.

```
SET TIMEDATE DATE("1 June 1984")
```

When programming without CLOCK, the user-defined function FNDate can be used.

```
SET TIMEDATE FNDate("1 June 1984")
```

Both of these statements are equivalent to the following statement.

```
SET TIMEDATE 2.113216992E+11
```

The DATE and FNDate functions convert the accompanying string (or string expression) into the numeric value needed to set the clock. To read the clock, the DATE$ and FNDate$ functions format the clock's value as the day, month, and year. For example, the following line will print the date.

```
PRINT DATE$(TIMEDATE)
```

Prints: `1 Jun 1984`

Programs that need to run without can use the following user-defined functions appended to the end of the program. These functions simulate the DATE and DATE$ keywords available in CLOCK. The algorithm is valid over the entire range of the clock.

Note the following functions are restricted to values the clock will accept, the DATE and DATE$ functions available with CLOCK allow a much wider range of values (including negative years).

```
10 Show_date:   DISP FNDate$(TIMEDATE)
20              GOTO Show_date
30              END
40    !
50    ! While the program is running, type:
60    ! SET TIMEDATE FNDATE("1 JAN 82")   <EXECUTE>
70    !
80    !*********************************************************
90    !
100   DEF FNDate$(Seconds) ! Given 'SECONDS' Return 'dd mmm yyyy'
110     !
120     DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
130     DIM Month$(1:12)[3]
140     READ Month$(*)
150     !
160     Julian=Seconds DIV 86400-1721119
170     Year=(4*Julian-1) DIV 146097
180     Julian=(4*Julian-1) MOD 146097
190     Day=Julian DIV 4
200     Julian=(4*Day+3) DIV 1461
210     Day=(4*Day+3) MOD 1461
220     Day=(Day+4) DIV 4
230     Month=(5*Day-3) DIV 153      ! Month
240     Day=(5*Day-3) MOD 153
```

```
250     Day=(Day+5) DIV 5             ! Day
260     Year=100*Year+Julian          ! Year
270     IF Month<10 THEN
280       Month=Month+3
290     ELSE
300       Month=Month-9
310       Year=Year+1
320     END IF
330     OUTPUT D$ USING "#,ZZ,X,3A,X,4Z";Day,Month$(Month),Year
340     RETURN D$
350   FNEND
360   !
370   DEF FNDate(Dmy$) ! Given 'dd mmm yyyy' Return 'SECONDS'
380     !
390     DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
400     DIM Month$(1:12)[3]
410     READ Month$(*)
420     !
430     ON ERROR GOTO Err
440     I$=Dmy$&"     "
450     ENTER I$ USING "DD,4A,5D";Day,M$,Year
460     IF Year<100 THEN Year=Year+1900
470     FOR I=1 TO 12
480     IF POS(M$,Month$(I)) THEN Month=I
490     NEXT I
500     IF Month=0 THEN Err
510     IF Month>2 THEN
520       Month=Month-3
530     ELSE
540       Month=Month+9
550       Year=Year-1
560     END IF
570     Century=Year DIV 100
580     Remainder=Year MOD 100
590     Julian=146097*Century DIV 4+1461*Remainder DIV 4+(153*Month+2) DIV 5+Day
+1721119
600     Julian=Julian*86400
610     IF Julian<2.08662912E+11 OR Julian>=2.143252224E+11 THEN Err
620     RETURN Julian ! Return Julian date in SECONDS
630 Err:OFF ERROR       ! ERROR in input.
640     RETURN TIMEDATE ! Return current date.
650   FNEND
```

Store the program in a file named "FUNDATE". Later the functions can be appended to other programs by the LOADSUB statement.

The functions FNDate$ and FNDate format the date as "DD MMM YYYY", where DD is the day of the month, MMM is the first three letters of the month, and YYYY is the year. The function FNDate will accept the last two digits of the year. See line 460. Note that the FNDate function requires two digits for the day, while the DATE function does not.

Different formats require only slight modification. By changing the following lines, the date is formatted as "MM/DD/YYYY".

```
330 OUTPUT D$ USING "#,2D,A,2D,A,2D";Month;"/";Day;"/";Year

450 ENTER I$ USING "#,ZZ,K";Month;Day;Year
```

European date format is obtained by swapping the month and day in the above statements. When changing the format, be sure to switch both functions.

If the all numeric format is chosen, delete the three lines in each function that load the array with the month mnemonics.

Using the Routines

The following statements summarize setting and displaying the clock.

```
SET TIMEDATE FNDate("12 DEC 1981") + FNTime("13:44:15")

SET TIME FNTime("8:30:00")

PRINT FNTime$(TIMEDATE)

DISP FNDate$(TIMEDATE)
```

It is important to note that `SET TIMEDATE` expects a date and time while the DATE function and the user-defined function FNDate return only a date. This effectively sets the clock to midnight of the date specified.

To keep the functions short, minimal parameter checking is performed. Additional checking may be incorporated within the functions or within the calling context. If FNDate or FNTime cannot correctly decode the input, the current value of the clock is returned.

The date and time functions can be used with the following program shell to provide a "friendly" interface to the clock.

```
10    ! PROGRAM SHELL FOR SETTING TIME AND DATE.
20    !
30    ! REQUIRES THE TIME AND DATE FUNCTIONS.
40    !
50    DIM Day$(0:6)[9]
60    DATA Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday
70    READ Day$(*)
80    !
90    ON ERROR GOTO Nofun      ! Test if functions
100   Dmy$=FNDate$(TIMEDATE)   ! have been loaded
110   Hms$=FNTime$(TIMEDATE)
120   OFF ERROR
130 Main:                      ! Get NEW date
140   GOSUB Clear_screen
150   F$=CHR$(255)&CHR$(72)
160   !
170   PRINT TABXY(1,14);"Enter the date, and press CONTINUE."
180   OUTPUT 2 USING "#,11A,2A";Dmy$,F$
190   !
200   INPUT Dmy$               ! WAIT for INPUT
210   !
220   ENTER Dmy$ USING "2D,4A,5D";D,M$,Y
230   GOSUB Clear_screen
240   !
250   PRINT TABXY(1,14);"Enter the time of day and press CONTINUE"
260   OUTPUT 2 USING "#,11A,2A";Hms$,F$
270   INPUT Hms$
280   ENTER Dmy$ USING "2D,4A,5D";D,M$,Y
290   !
300   SET TIMEDATE FNDate(Dmy$)+FNTime(Hms$)
310   !
320   GOSUB Clear_screen
330   W=(TIMEDATE DIV 86400) MOD 7 ! Day of week
340   PRINT TABXY(1,1);"The clock has been set to:"
350   PRINT TABXY(1,3);Day$(W);" ";Dmy$;"  ";FNTime$(TIMEDATE)
360   GOTO Quit
```

```
370   !
380   ! ************ SUBROUTINES ************
390   !
400 Clear_screen:OUTPUT 2 USING "#,B";255,75
410   RETURN
420 Nofun:PRINT "The TIME & DATE FUNCTIONS must be appended,"
430   PRINT "(via LOADSUB) before program will work."
440 Quit:END
450   !
460   ! ************* FUNCTIONS **************
470   !
480   ! append time and date functions here
```

The program tests to see if the functions have been loaded by trying to use them. If they are not loaded the program ends with an error message. With CLOCK, this program can still be used. Replace the calls to the user-defined functions with the appropriate keywords. The error trapping can then be deleted.

To append the functions, execute the following statements while the demonstration program is in memory.

```
LOADSUB ALL FROM "FUNDATE"

LOADSUB ALL FROM "FUNTIME"
```

Examine the program to be sure the functions have been loaded.

The program will prompt for the date and time, then set the clock accordingly. A program such as this may be used as the system start-up program for applications requiring the date or time.

Day of the Week

An advantage of Julian dates is the simplicity of finding the day of the week. `TIMEDATE DIV 86400 MOD 7` returns a number which represents the day of the week. Monday is represented by zero (0), and the numbering continues through the week to Sunday which is represented by six (6). See the previous program for an example of using this routine.

Days Between Two Dates

The number of days between two dates is easily calculated as the following program demonstrates.

```
10    ! Days between two dates
20    INPUT "ENTER THE FIRST DATE (DD MMM YYYY)",D1$
30    INPUT "ENTER THE SECOND DATE (DD MMM YYYY)",D2$
40    Days=(DATE(D2$)-DATE(D1$)) DIV 86400
50    DISP Days;"days between '";D1$;"' and '";D2$;"'"
60    END
```

Interval Timing

Timing a single event of short duration is quite simple.

```
10    T0=TIMEDATE          ! Start
20    FOR J=1 TO 5555
30    !
40    NEXT J
50    T1=TIMEDATE          ! Finish
60    !
70    PRINT "It took";DROUND(T1-T0,3);"seconds"
80    END
```

Programs can and should be written so that they do not change the setting of the clock. A short program, which simulates a stopwatch, allows interval timing without changing the clock.

```
10    ! Program: STOPWATCH
20    ! Interval timing without changing the clock
30    ON KEY 5 LABEL " START " GOTO Start
40    ON KEY 6 LABEL " STOP  " GOTO Hold
50    ON KEY 7 LABEL " RESET " GOTO Reset
60    ON KEY 8 LABEL "  LAP  " GOSUB Lap
70    !
80 Reset:PRINT CHR$(12)              ! form-feed
90    H=0                            ! Set all
100   M=0                            !   to
110   S=0                            !  zero.
120   !
130 Hold:DISP TAB(9);H;":";M;":";S   ! Wait til
140   GOTO Hold                      !  keypress
150   !
160 Lap:PRINT H;":";M;":";S          ! Print lap
170   RETURN
180   !
190 Start:Z=3600*H+60*M+S-TIMEDATE   ! Elapsed-
200 Loop:T=(TIMEDATE+Z) MOD 86400    !      time
210   T=INT(T*100)/100               ! .01 sec.
220   H=T DIV 3600                   ! Hours
230   M=T MOD 3600 DIV 60            ! Minutes
240   S=T MOD 60                     ! Seconds
250   DISP TAB(9);H;":";M;":";S      ! Show time
260   GOTO Loop                      ! Do again
270   END
```

Accuracy

The computer's crystal controlled clock maintains the time to one one-hundredth of a second, however when timing events over a long period of time, the accuracy of the crystal must be considered. Under normal operating conditions, this error will be less than ±5 seconds per day.

If the powerfail option is installed, accuracy is improved. Under normal operating conditions, the powerfail clock error is less than ±2.5 seconds per day.

Powerfail Protection

Without the powerfail option, each time power is applied the date is initialized to March 1, 1900 (2.08662912E+11 seconds). If powerfail is installed, a second clock located on the powerfail board takes over the timekeeping function and maintains the date and time when the computer is turned off. (The powerfail option is not available on the Model 216.)

Branching on Clock Events

Several additional branching statements, available with CLOCK, allow end-of-statement branches to be triggered according to the real-time clock's value.

- ON TIME enables a branch to be taken when the clock reaches a specified time of day.
- ON DELAY enables a branch to be taken after a specified number of seconds has elapsed.
- ON CYCLE enables a recurring branch to be taken with each passage of a specified number of seconds.

The specified time can range from 0.01 thru 167772.15 seconds for the ON CYCLE and ON DELAY statements and 0 thru 86399.99 seconds for ON TIME. The value specified with ON TIME indicates the time of day (in seconds past midnight) for the branch to occur.

Each of these statements has a corresponding statement to cancel the branch (OFF TIME, OFF DELAY, and OFF CYCLE). A statement is also canceled by executing another ON TIME, ON DELAY, or ON CYCLE statement.

All of the statements use the internal real-time clock. Care should be taken to avoid writing programs that could change the clock's setting during execution. Since only one resource is dedicated to each statement, certain restrictions apply to the use of these statements.

Cycles and Delays

Both the ON CYCLE and ON DELAY statements enable a branch to be taken as soon as the specified number of seconds has elapsed. ON CYCLE remains in effect, re-enabling a branch with each passage of time. For example:

```
10     ON CYCLE 1 GOSUB Five   ! Print 5 random numbers every second.
20     ON DELAY 6 GOTO Quit    ! After 6 seconds quit.
30     !
40 T:  DISP TIME$(TIMEDATE)    ! Show the time.
50     GOTO T
60     !
70 Five:FOR I=1 TO 5
80       PRINT RND;
90     NEXT I
100    PRINT
110    RETURN
120    !
130 Quit:END
```

The program will print five random numbers every second for six seconds and then stop.

Only one ON CYCLE and one ON DELAY statement can be active in a program context. Executing a second ON CYCLE or ON DELAY statement in the same program context deactivates the first ON CYCLE or ON DELAY statement. If a branch is missed, due to priority restrictions or execution of a subprogram, the event is logged and the branch will be taken when the restriction is removed or the original context is restored. If an active ON CYCLE or ON DELAY statement gets canceled in an alternate context (subprogram) the branch is restored when execution returns to the defining context. (See Branching Restrictions for more information about this).

Time of Day

The ON TIME statement allows you to define and enable a branch to be taken when the clock reaches a specified time of day, where time of day is expressed as seconds past midnight. Using the TIME function simplifies setting an ON TIME statement by allowing a formatted time of day to be used. For example:

```
ON TIME TIME("11:30") GOTO Lunch
```

Typically, the ON TIME statement is used to cause a branch at a specified time of day. By adding an offset to the current clock value, the ON TIME statement can be used as an interval timer. In the following example, both ON DELAY and ON TIME are used as interval timers.

```
10    ON DELAY 5 GOSUB Takeoff                          ! delay 5 seconds
20    ON TIME (TIMEDATE+10) MOD 86400 GOSUB Touchdown ! delay 10 seconds
30    PRINT "STARTING... ",TIME$(TIMEDATE)
40 Clock:DISP TIME$(TIMEDATE)
50    GOTO Clock
60    !
70 Takeoff:PRINT "TAKEOFF at ",TIME$(TIMEDATE)
80    RETURN
90 Touchdown:PRINT "TOUCHDOWN at ",TIME$(TIMEDATE)
100   RETURN
110   END
```

The starting time is printed when the program is executed. Five seconds later the first subroutine is executed. Ten seconds after the program starts, the second subroutine is executed.

Only one ON TIME statement can be active in a program context. If a branch is missed, due to priority restrictions or execution of a subprogram, the event is logged and the branch will be taken when the restriction is removed or the original context is restored. If an active ON TIME statement gets canceled in an alternate context (subprogram) the branch is restored when execution returns to the defining context. (See Branching Restrictions for more information about this).

Due to the "match an exact time" nature of the ON TIME statement, if the specified time occurs when the statement is temporarily canceled (by an OFF TIME in an alternate context), no branch will be taken when the defining context is restored.

Priority Restrictions

A priority can be assigned to the branch defined by ON CYCLE, ON DELAY, and ON TIME. For example:

```
ON CYCLE Seconds,Priority GOTO Label
```

If the system priority is higher than the branch priority at the time specified for the branch, the event will be logged but the branch will not be taken until the system priority falls below the branch priority. An example program follows.

```
10    COM Start
20    P=0
30 Up:P=P+1
40    IF P>15 THEN Quit            ! Priority from 1 thru 15
50    PRINT
60    PRINT "Priority:";P;
70    Start=TIMEDATE               ! Save the start-time for subprogram.
80    ON CYCLE 1,P RECOVER Up      ! New priority every second if not Busy.
90    ON DELAY .5,6 CALL Busy      ! DELAY overrides CYCLE until priority
100                                !                  (P) is greater than 6.
110 W:GOTO W
120 Quit:END
130   !----------------- SUB has priority of 6 --------------------
140   SUB Busy
150     COM Start
160     PRINT "SUB";
170     WHILE I<10
180       IF TIMEDATE>Start+1 THEN  ! Has ON CYCLE time been exceeded?
190         PRINT "*";              ! YES (only prints if Priority<7)
200       ELSE
210         PRINT ".";              ! NO
220       END IF
230       I=I+1                     ! Loop ten times
240       WAIT .1
250     END WHILE
260     PRINT "DONE";
270   SUBEND
```

Once the priority assigned to the ON CYCLE statement is greater than the priority assigned to the ON DELAY statement (6) the subprogram will be interrupted. After running the program, change line 80 in the above program to the following:

```
80    ON CYCLE 1,P GOTO Up
```

Running the new version of the program will show that GOTO (or GOSUB) will not interrupt a subprogram regardless of the assigned priority. The branch will be logged but not taken until execution returns to the main program. If you write a program that makes extensive use of subprograms and branching statements, use CALL and RECOVER to insure proper operation.

Branching Restrictions

Certain restrictions apply to the use of ON TIME, ON CYCLE, and ON DELAY because only one resource is dedicated to each statement. Assuming an active branch has been defined in the main program, execution of a subprogram which sets up a new branch, will cause the loss of the original time. When the main program context is restored, the original branch will be restored, but at the time defined in the subprogram. The following program will illustrate this effect.

```
10     COM Counter
20     Counter=0
30     GINIT
40     GRID 1,1                ! Fill graphics raster with grid.
50     DISP Counter
60     ON CYCLE 2 CALL Flash   ! Flash graphics every 2 seconds.
70 W:  GOTO W
80     END
90     !----------------- SUB to flash graphics raster -------------
100    SUB Flash
110      COM Counter
120      GRAPHICS ON
130      Counter=Counter+1
140      DISP Counter
150      IF Counter=5 THEN          ! Change CYCLE value during fifth CALL.
160        ON CYCLE .1,2 CALL Quit  ! New value (.1) will replace old (2).
170                                 ! Flash will end before Quit gets called.
180      END IF
190      GRAPHICS OFF
200    SUBEND
210    !----------------- SUB that won't get called ----------------
220    SUB Quit
230      PRINT "PROGRAM HAS STOPPED"
240      STOP
250    SUBEND
```

The program starts out by flashing the graphics raster on and off every two seconds. When the subprogram's ON CYCLE statement is activated during the fifth call to the subprogram, the new value (0.1 second) replaces the old value (2.0 seconds) and the program begins flashing the graphics raster at the new rate. Note that the branch to the second subprogram (Quit) is not executed because the first subprogram is finished before the specified time. To see the second subprogram execute, insert the following line.

```
191      WAIT 1
```

The delay caused by the WAIT statement allows the subprogram's ON CYCLE statement to branch to the second subprogram and stop execution.

If an active branch defined in the main program is canceled in a subprogram (by OFF TIME, OFF DELAY, or OFF CYCLE) any branch missed during the execution of the subprogram will be lost. When the context containing the original statement is restored, the branch will be reactivated and processing will continue as if no branch was missed.

```
10    ON DELAY 1 GOTO Done        !  GOTO "Done" in one second.
20    CALL Busy                   ! Call to "Busy" takes two seconds.
30    !
40    PRINT "THIS WON'T BE PRINTED UNLESS BRANCH IS CANCELED BY THE SUB"
50    !
60 Done:PRINT "THIS LINE WILL BE PRINTED EVERY TIME"
70    END
80    ! ------------------------------------------------------------
90    SUB Busy
100     WAIT 2
110   ! OFF DELAY  !  RUN then remove the "!" on this line and RUN again.
120   SUBEND
```

By removing the comment symbol (!) from the beginning of line 110, the OFF DELAY statement will be executed causing any branch that has already been logged to be canceled and allow line 40 to be printed.

Since branches only occur at the end of a line, no branch can be taken during an INPUT or LINPUT statement. The following program shows a method of monitoring the keyboard without preventing branches to be taken.

```
10    ON KBD GOTO Yes           ! If key is pressed go get new value.
20    ON DELAY 3 GOTO Gone      ! If no keypress in 3 seconds use defaults
30    DISP "PRESS A KEY"
40 W: GOTO W                    ! Wait here until keypress or end of delay.
50    !
60 Yes:OFF DELAY                ! Someone is there.
70    OFF KBD
80    LINPUT "NEW VALUE?",A$
90    DISP "USING",A$
100   GOTO More
110   !
120 Gone:DISP                   ! Nobody there.
130   DISP "USING DEFAULTS"
140   !
150 More:WAIT 2
160   DISP "Program continues...."
170   END
```

The program waits a few seconds for a response. Processing continues with default values if no key is pressed. Pressing a key will cause the program to accept the new information.

Communicating with the Operator

Chapter 10

Introduction

It is very unlikely that a computer could perform useful work without receiving input. Much of that input is from electronic devices: instruments, mass storage devices, other computers, and so on. Because a computer is an electronic device, it is very good at these tasks. There are also times when the computer's input must come from the human sitting in front of the computer.

Good human interfaces do not happen without some effort from the programmer. In many programs, at least one fourth of the code is dedicated to human interface. It is not unusual to use one half of a good program for operator interaction, error trapping, explanatory messages, etc. Obviously, these estimates depend upon many factors, like the task being performed and the intended operators. If you are the only person who uses a program, that program may not need a quality human interface. However, the demands for a good human interface rise greatly if a program is used by many people with different backgrounds. When the intended users do not understand computers, your program must be very skillfully written so that it does not intimidate the operator or make great demands.

This chapter introduces two of the elements of a human interface: displaying text for the operator to read and accepting operator input from the keyboard. These are certainly not the only elements in a human interface. A good human interface can involve the placement of hardware, use of graphic and voice communication, data base management, artificial intelligence theories, and much more. However, you must begin somewhere. Despite the incredible technology growing up around them, many programmers fail at the basic task of sending and receiving text. Hopefully, the hints in this chapter will help your present programs and whet your appetite for more eleborate improvements in future programs.

Displaying and Prompting

One of the simpler things to do for the operator is to display an explanation of what is happening or what is expected. In the early days of computers, memory was a scarce and expensive resource. Old-time programmers were encouraged to use as little memory as possible. It seemed as though there was a contest to see who could put the most information into a 32-character message. Please realize that those days are over. For example, in the Model 236, there is no significant restriction on program size, the standard machine is shipped with over a half-million characters of memory, and there are 18 lines of 80 characters visible at all times on the CRT. If you are sending your operator tiny, cryptic 32-character messages, you are making an unnecessary mistake.

Giving instructions to the operator can be viewed as two basic steps:

1. Clear the CRT.
2. Use as much of the CRT as necessary to give readable instructions.

Determining Screen Width

The first step in dealing with the CRT is finding out the size of the CRT. Programs written for Series 200 computers can be used on either 50, 80 or 128-column displays. There is a CRT status register that contains the width of the CRT. If you are developing programs that will be transported between models, status register 9 will be very helpful to you. The screen width is useful in centering displays, labeling softkeys, formatting tabular data, and other display tasks. The following statement places the screen width in a variable called `Crt_width`.

```
STATUS 1,9;Crt_width
```

There is also a SYSTEM$ function that returns useful information about the CRT. The specifier "CRT ID" returns a string containing (amoung other things) the screen width and availability of highlights and graphics. The following example shows one method of determining the screen width with SYSTEM$.

```
120  Test$=SYSTEM$("CRT ID")
130  Screen=VAL(Test$[3,6])
```

Clearing the CRT

It is embarassing to the programmer and confusing to the operator when two or more displays combine in an unplanned manner. The culprits are "left-over" alpha and "left-over" graphics. Left-over alpha can occur for a number of reasons:

- The operator may have used the knob or cursor-control keys to scroll text from the off-screen buffer.
- With TABXY, the PRINT statement overwrites any old characters on a line with new characters. However, if the old text is longer than the new text, the end of the old line remains visible. Therefore, the following sequence does **not** print three blank lines. It just moves the print position. Any old lines will still be on the screen.

```
100  PRINT
110  PRINT
120  PRINT
```

- If the PRINTALL mode is on, all interactions on the display line and keyboard input line are sent to the output area.

Turning Off Unwanted Modes

There are several modes that affect the appearance of the CRT. These are very useful for certain purposes, but generally undesirable for the display of simple text. Graphics is an obvious example. Left-over graphics can be removed by the following statement on non-bit mapped displays.

```
GRAPHICS OFF
```

The PRINTALL mode is canceled by writing a zero in the PRINTALL control register. This is keyboard register 1, so it has an interface select code of 2. The following statement turns off the PRINTALL mode.

```
CONTROL 2,1;0
```

The DISPLAY FUNCTIONS mode can make a display look sloppy. This is CRT register 4, so it has an interface select code of 1. The following statement turns off the DISPLAY FUNCTIONS mode.

```
CONTROL 1,4;0
```

Printing Blank Lines

To print a line that is blank is a different operation from sending only an end-of-line sequence. A `PRINT` statement with no parameters simply sends an end-of-line sequence. If the print position is at the start of a blank line when `PRINT` is executed, that line remains blank. However, if there is text on that line, the text remains. This is not to say that it is "wrong" to use `PRINT` with no parameters. It just means that you cannot guarantee the output of a blank line by using `PRINT` with no parameters.

To print a blank line, blanks must be printed. One of the most convenient ways to send a line full of blanks is the `TAB` function. Here is a sequence that prints three blank lines:

```
100  STATUS 1,9;Screen
110  PRINT TAB(Screen)
120  PRINT TAB(Screen)
130  PRINT TAB(Screen)
```

Using OUTPUT KBD...

The PRINT statement does not provide functions like "home" and "clear", but the keyboard drivers do. Note that this is true even for keyboards which do not have a "home" key. These functions are invoked by sending a "Non-ASCII Key Sequence" to interface select code 2 (KBD). Sending characters with OUTPUT KBD is like telling the computer to press its own keys. Although some techniques that use the keyboard buffer are very complex (see Chapter 9 of *BASIC Interfacing Techniques*), controlling the CRT output area is simple.

To see how this technique works, let's use the CLR SCR Key (Clear display on HP 46020A) as an example. Open your *BASIC Language Reference* to the end of the "Useful Tables" section. Find the heading "Second Byte of Non-ASCII Key Sequences (Numeric)". Locate the CLR SCR key on your computer. It is a shifted CLR LN (on non-HP 46020A keyboards). The number in that position is 75. Like the heading says, that is the value of the **second** byte of a sequence. The first byte always has a value of 255 for non-ASCII keys. Therefore, to "press" the CLR SCR key, send the bytes 255 and 75 to the keyboard; interface select code 2.

```
1480     ON KEY 8 LABEL " Roman  " GOTO Numbers
1490     ON KEY 9 LABEL " START  " GOTO Begin
1500   ELSE                                ! Use long labels
1510     ON KEY 5 LABEL "Command Delim " GOTO Command
1520     ON KEY 6 LABEL " Select Drive " GOTO Drive
1530     ON KEY 7 LABEL " Stand. Fmt.? " GOTO Standard
1540     ON KEY 8 LABEL "Roman Numeral?" GOTO Numbers
1550     ON KEY 9 LABEL " START PRINT  " GOTO Begin
1560   END IF
1570   ON KEY 0 GOTO Not_used            ! Turn off unused keys
1580   ON KEY 1 GOTO Not_used
1590   ON KEY 2 GOTO Not_used
1600   ON KEY 3 GOTO Not_used
1610   ON KEY 4 GOTO Not_used
1620   !
1630 Spin:  GOTO Spin                    ! Wait for softkey interrupt
1640   !
1650 Not_used:  !
1660   BEEP 300,.1                       ! Feedback for unused keys
1670   GOTO Spin
1680   !
1690 Command:  !
1700   IF Cmd$="\" THEN                  ! Choose command delimiter
1710     Cmd$="^"
1720   ELSE
1730     Cmd$="\"
1740   END IF
1750   GOTO Menu
1760   !
1770 Drive:  !
1780   IF Disc$="RIGHT" THEN             ! Choose text source
1790     MASS STORAGE IS ":INTERNAL,4,1"
1800     Disc$="LEFT "
1810   ELSE
1820     MASS STORAGE IS ":INTERNAL,4,0"
1830     Disc$="RIGHT"
1840   END IF
1850   GOTO Menu
1860   !
1870 Standard:  !
1880   IF Std_fmt THEN                   ! Choose text format
1890     Std_fmt=0
1900   ELSE
1910     Std_fmt=1
1920   END IF
1930   GOTO Menu
1940   !
1950 Numbers:  !
1960   IF Roman THEN                     ! Choose numeral type
1970     Roman=0
1980   ELSE
1990     Roman=1
2000   END IF
2010   GOTO Menu
2020   !
2030 Begin:  !
2040   OUTPUT 2;Clear$;                  ! Clear CRT
2050   OFF KEY                           ! Remove selection menu
2060   !
2070   ! Program continues here when user presses "START"
2080   !
```

The program uses softkeys 5 through 9. If you have an HP 46020A keyboard, your softkeys are labeled 1 through 8. You can modify the program to use the softkeys most useful for your applications.

It is always good programming practice to declare all variables. The first two lines do this. Next, the variables are given their starting values. Initialization is completed by turning off unwanted modes and clearing the CRT.

The section at "Menu" displays a description and current status for each menu item. This example shows some of the parameters that might be used by a simple text-printing program. The items used are representative only. A real text formatter would have many more parameters (all the more reason to present them clearly). The operator can choose the following:

- Back-slash or up-caret as a command delimiter
- Right or left disc drive for the source of the text
- Standard or alternate format for the text
- Page numbering with Arabic or Roman numerals

Notice some important aspects of this menu. All items have default values and all defaults are visible simultaneously. This is very important. It is irritating and confusing when an operator must answer question after question to get a program to begin. It is far better to show the default environment and allow a single keypress to start the program if the defaults are acceptable. If any defaults need to be changed, the operator changes only those items he wants to change. He can press "START" at any time, and in this simple case, never answers any questions. The operator wants a printout, not a game of "20 questions".

The current state of all items is displayed in a form that is meaningful to the operator. It is reasonably safe to assume that all operators know what "RIGHT" and "LEFT" mean. Very few would have any idea what ":INTERNAL,4,1" means. Programmers need to learn about concepts like "mass storage unit specifier". Operators shouldn't be bothered by such things. Likewise, don't expect anyone to answer "1" or "0" to a question that should be answered "YES" or "NO".

A more technical aspect of this menu is the method used to update the display. Since the scrolling keys are on one side of the softkeys and the knob is on the other side, it is reasonable to assume that the operator might accidentally move the display out of place. One way to correct this would be to start each display update with a "clear screen" sequence. This guarantees the state of the CRT and the print position. Unfortunately, it also causes a very undesirable "blinking off" of the display each time a key is pressed. A constantly disappearing menu is very distracting.

The objective is to give the impression that nothing changed except the selected item. Therefore, the "clear" sequence is sent before the first display only. Subsequent updates use a "home" sequence to ensure the position of the text, and a TABXY to set the print position. As a result, the new menu is written on top of the old menu. (The same visual effect could be achieved by using individual TABXY functions to access each item display, but that is a more difficult program to write.)

Since the old display is overwritten each time, it is important to erase all unneeded characters. Notice that the "NO" displays are padded with a trailing blank to erase the "S" left over from "YES". This technique can be extended to clear old displays of unknown length. The following example displays a number and erases any remaining digits from the old number. The variable `Screen` contains the screen width.

```
1300  PRINT Value;TAB(Screen)
```

The function uses a local copy of the string just in case you need the actual input for some other purpose in the main program. The response is trimmed and placed in uppercase. Then the first letter is tested. Four cases are identified: the answer was "Y" (for yes), the answer was "N" (for no), no answer was given, or the answer was not recognized.

```
2000  DEF FNYes(X$)
2010    DIM Temp$[1]
2020    Temp$[1,1]=TRIM$(X$)
2030    SELECT Temp$
2040    CASE "Y","y"
2050      RETURN 1
2060    CASE "N","n"
2070      RETURN 0
2080    CASE " "
2090      RETURN -1
2100    CASE ELSE
2110      RETURN -2
2120    END SELECT
2130  FNEND
```

As mentioned previously, every question should have a default answer. The default answer for a yes/no question depends greatly upon the nature of the question. If you are asking the operator for permission to use standard, reasonable parameters for an operation, then "yes" is a helpful default. If you are asking for permission to initialize a disc and destroy all files, then the default answer had better be "NO"! When a question or choice occurs more than once in a program, it is usually a good technique to use the operator's previous response as the default. Put yourself in the user's place and think about how the program should run.

To use this function to best advantage, the result must be tested thoughtfully. If the operator simply presses (CONTINUE), the result will be -1. Therefore, the default should be assumed if FNYes $= -1$. A "yes" answer is indicated by FNYes $= 1$; whereas a **non-negative** answer can be tested simply as `IF FNYes`. A **non-affirmative** answer is FNYes < 1. Any result less than zero is a noncommittal reply. Perhaps the default could be assumed for any negative result, or perhaps a negative result should cause the question to be repeated. The test `IF NOT FNYes` reveals a negative reply. As you can see, many shades of interpretation are possible.

Custom Interfaces

The ON KBD statement can be used to design very effective keyboard interfaces. However, these are usually very complex to program. In fact, using ON KBD to accept keyboard input, CRT highlights to display a cursor, and PRINT TABXY to position cursor and text is essentially writing a text editor. Unfortunately, programs of that magnitude are beyond the scope of this manual. Information about the applicable keywords can be found in the *BASIC Language Reference*, and some examples of CRT control and ON KBD can be found in Chapters 8 and 9 of *BASIC Interfacing Techniques*.

Error Handling

Chapter 11

Introduction

Most programs are subject to errors happening at run time, even if all the typographical/syntactical errors have been shaken out in the process of entering the program into the computer in the first place. There are three courses of action to take with respect to errors:

1. Try to prevent the error from happening in the first place
2. Once an error occurs, try to recover from it and continue execution
3. Do nothing — let the program roll over and die if an error happens

The last alternative, which may seem frivolous at first glance, is certainly the easiest to implement, and the nature of HP desktop computers is such that this is often a feasible choice. Upon encountering a run-time error, the computer will pause program execution, and display a message giving the error number and the line in which the error happened, and the programmer can then examine the program in light of this information and fix things up. The key word here is "programmer". If the person running the program is also the person who wrote the program, this approach works fine. If the person running the program did not write it, or worse yet, does not know how to program, some attempt should be made to prevent errors from happening in the first place, or to recover from errors and continue running.

Anticipating Operator Errors

When a programmer writes a program, he or she knows exactly what the program is expected to do, and what kinds of inputs make sense for the problem. Given this viewpoint, there is a strong tendency for the programmer not to take into account the possibility that other people using the program might **not** understand the boundary conditions. A programmer has no choice but to assume that every time a user has the opportunity to feed an input to a program, a mistake can be made and an error can be caused. If the programmer's outlook is noble, he or she will try to save the user from needless anguish and frustration. If the programmer's outlook is self-centered, he or she will try to keep from getting involved in future support problems. In either case, an effort must be made to make the program foolproof.

Boundary Conditions

A classic example of anticipating an operator error is the "division by zero" situation. An INPUT statement is used to get the value for a variable, and the variable is used as a divisor later in the program. If the operator should happen to enter a zero, accidentally or intentionally, the program crashes with an error 31. It is far better to be watching for an out-of-range input and respond gracefully. One method is shown in the following example.

```
100  INPUT "Miles traveled and total hours",Miles,Hours
110  IF Hours=0 THEN
120    BEEP
130    PRINT "Improper value entered for hours."
140    PRINT "Try again!"
150    GOTO 100
160  END IF
170  Mph=Miles/Hours
```

Consider another simple example of giving a user the choice of six colors for a certain bar graph. It might be preferable to have the user pick a number corresponding to the color he wished to choose instead of having to type in up to six characters. In this case, the program wouldn't have to check for each number, but rather it could use the logical comparators to check for an entire range:

```
4030  OUTPUT KBD;Clear$;  ! Clear the screen
4040  DATA GREEN,BLUE,RED,YELLOW,PURPLE,PINK
4050  ALLOCATE Colors$(1:6)[6]
4060  READ Colors$(*)
4070  FOR I=1 TO 6
4080     PRINT USING "DD,X,K";I,Colors$(I)
4090  NEXT I
4100 Ask: INPUT "Pick the number of a color",I
4110  IF I>=1 AND I<=6 THEN Valid_Color
4140  BEEP
4150  DISP "Invalid answer -- ";
4160  WAIT 1
4170  GOTO Ask
```

The above example needs a little extra safeguarding. I, the variable being input, should be declared to be an integer, since the only valid inputs are 1, 2, 3, 4, 5, and 6. An answer like "pick the 3.14th color listed" does not make sense.

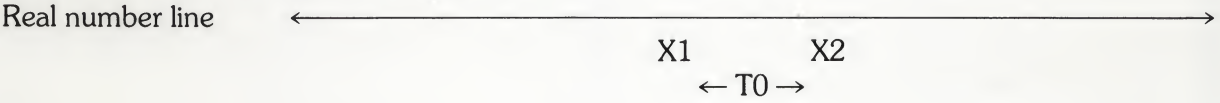
Real number boundaries are tested for in a manner similar to that of integers:

```
7010   INPUT "Enter the waveform's frequency (in KHz)",Frequency
7020   IF Frequency<=0 THEN 7010
7030   INPUT "Enter the amplitude (0-10 volts)",Amplitude
7040   IF Amplitude<0 OR Amplitude>10 THEN 7030
7050   INPUT "Enter the phase angle (in degrees)",Angle
7060   IF Angle<0 OR Angle>180 THEN 7050
7070   Angle=Angle*PI/180
```

REAL Numbers and Comparisons

A word of caution is in order about the use of the = comparator in conjunction with REAL (full precision) numbers. Numbers on this computer are stored in a binary form, which means that the information stored is not guaranteed to be an exact representation of a decimal number — but it will be real close! What this means is that a program should not use the = comparator in an IF statement where the comparison is being performed on REAL numbers. The comparison will yield a 'false' or '0' value if the two are different by even one bit, even though the two numbers might really be equal for all practical purposes.

There are two ways around this problem. The first is to try to state the comparison in terms of the <= or >= comparators. However if it's absolutely necessary to do an equality comparison with a pair of REAL numbers, then the second method must be used. This involves picking an error tolerance for how close to being equal the two numbers can be to satisfy the test.



So if the difference between two REAL numbers X1 and X2 is less than or equal to a tolerance T0, we'll say that X1 and X2 are "equal" to each other for all practical purposes. The value of T0 will depend upon the application, and must be chosen with care.

For an example, assume that we've picked a tolerance of 10^{-12} for comparing two REAL numbers for equality. The proper way to compare the two numbers would be:

```
950 IF ABS(X1-X2)<=1E-12 THEN Numbers_equal
960 ! Otherwise they're not equal
```

Another technique for comparing REAL values is to use the DROUND function. This is especially suited to applications where the data is known to have a certain number of significant digits. For more details on binary representations of decimal numbers, refer to Chapter 4.

Error Trapping

Despite the programmer's best efforts at screening the user's inputs in order to avoid errors, sometimes an error will still happen. It is still possible to recover from run time errors, provided the programmer predicts the places where errors are most likely to happen.

ON/OFF ERROR

The ON ERROR command sets up a branching condition which will be taken any time a recoverable error is encountered at run time. The branching action taken may be either GOTO, GOSUB, CALL, or RECOVER. GOTO and GOSUB are purely local in scope — that is, they are active only within the context in which the ON ERROR is declared. CALL and RECOVER are global in scope — after the ON ERROR is set up, the CALL or RECOVER will be executed any time an error occurs, regardless of subprogram environment.

When an ON ERROR statement is executed, the language system will make sure that the specified line or subprogram exists in memory before the program will proceed. If ON ERROR GOTO/GOSUB/RECOVER are specified, then the line identifier must exist in the current context. If an ON ERROR CALL is given, then the specified subprogram must currently be in memory. In either case, if the system can't find the given line, an error 49 is issued.

If either ON ERROR GOSUB or ON ERROR CALL are used and an error occurs, the specified branch will take place, and when the RETURN or SUBEXIT is executed, then program execution will resume at the line which caused the error, and an attempt will be made to execute the line again.
ON ERROR has a priority of 16, which means that it will always take priority over any other ON <event> since the highest user-specifiable priority is 15.

The OFF ERROR statement will cancel the effects of the ON ERROR statement, and no branching will take place if an error is encountered.

The DISABLE statement has no effect on ON ERROR branching.

ERRN/ERRL/ERRM$

ERRN is a function which returns the error number which caused the branch to be taken. ERRN is a global function, meaning it can be used from the main program or from any subprogram, and it will always return the number of the most recent error.

ERRM$ is a string function which returns the text of the error which caused the branch to be taken.

ERRL is a function which is used to find the line in which the error was encountered. ERRL is a boolean function. The program feeds it a line identifier, and either a 1 or a 0 is returned, depending upon whether or not the specified identifier indicates the line which caused the error. ERRL is a local function, which means it can only be used in the same environment as the line which caused the error. This implies that ERRL cannot be used in conjunction with ON ERROR CALL, and that it can be used with ON ERROR GOTO and ON ERROR GOSUB. ERRL can be used with ON ERROR RECOVER only if the error did not occur in a subprogram which was called by the environment which set up the ON ERROR RECOVER.

The ERRL function will accept either a line number or a line label.

```
1140   DISP ERRL(710)
910    IF ERRL(Compute) THEN Fix_compute
```

ON ERROR GOSUB

The ON ERROR GOSUB statement should only be used when you can guarantee that the problem causing the error can be fixed and the line can be re-executed safely. Remember that if the action taken in the error service routine is not sufficient to correct the problem, the program will dive into an infinite loop. Every time an error occurs, a GOSUB will cause a branch to the error service routine which will RETURN execution to the line causing the error.

When an error triggers a branch as a result of an ON ERROR GOSUB statement being active, system priority is set at the highest possible level (16) until the RETURN statement is executed, at which point the system priority is restored to the value it was when the error happened.

```
100    Radical=B*B-4*A*C
110    Imaginary=0
120    ON ERROR GOSUB Esr
130    Partial=SQR(Radical)
140    OFF ERROR
.
.
.
350 Esr: IF ERRN=30 THEN
360         Imaginary=1
370         Radical=ABS(Radical)
380      ELSE
390         BEEP
400         DISP "Unexpected Error ("&ERRN&")"
410         PAUSE
420      END IF
430      RETURN
```

ON ERROR GOTO

The ON ERROR GOTO statement is generally more useful than ON ERROR GOSUB, especially if you are trying to service more than one error condition. The only advantage that ON ERROR GOSUB has over ON ERROR GOTO is that system priority is maintained at the highest possible level until the error subroutine is finished.

By using the ON ERROR GOTO statement, the same error service routine can be used to service all the error conditions in a given context. By testing both the ERRN (what went wrong) and the ERRL (where it went wrong) functions, proper recovery procedures can be taken.

```
10    RESTORE
20    PRINT
30    PRINT
40    PRINT "Coefficients of quadratic equation A"
50    DATA 0,0,0
60    READ A,B,C
70    Maxreal=1.79769313486231E+308
80    Overflow=0
90 Coefficients:    !
100   INPUT "A?",A
110   IF A=0 THEN
120      DISP "Must be quadratic"
130      WAIT 1.5
140      GOTO Coefficients
150   END IF
160   PRINT "A=";A
170   INPUT "B?",B
180   PRINT "B=";B
190   INPUT "C?",C
200   PRINT "C=";C
210 Compute_roots:   !
220   ON ERROR GOTO Esr
230   Imaginary=0
240   Part1=-B/(2.*A)
250   Part2=SQR(B*B-4*A*C)/(2.*A)
260   IF NOT Imaginary THEN
270      Root1=Part1+Part2
280      Root2=Part1-Part2
290   END IF
300   OFF ERROR
310 Print_roots:  !
320   IF Imaginary=0 THEN
330      PRINT "Root 1 =";Root1
340      PRINT "Root 2 =";Root2
350   ELSE
360      PRINT "Root 1 =";Part1;" +";Part2;" i"
370      PRINT "Root 2 =";Part1;" -";Part2;" i"
380   END IF
390   IF Overflow THEN PRINT "OVERFLOW"
400   STOP
410 Esr:   !
420   IF ERRN=30 THEN     ! SQR OF NEGATIVE NUMBER
430      Part2=SQR(ABS(B*B-4*A*C))/(2*A)
440      Imaginary=1
450      Branch=1
460      GOTO 270
470   ELSE
480      IF ERRN=22 THEN  ! REAL OVERFLOW
490         Overflow=1
500         SELECT 1
510         CASE ERRL(240)
520            Part1=SGN(B)*SGN(A)*Maxreal
530            Branch=2
540         CASE ERRL(250)
550            Part2=Maxreal
560            Branch=3
580         CASE ERRL(270)
590            Root1=Maxreal*SGN(Part1)
600            Branch=4
620         CASE ERRL(280)
630            Root2=Maxreal*SGN(Part1)
640            Branch=5
660            PRINT "UNEXPECTED OVERFLOW"
670            Branch=6
680         CASE ELSE
690            DISP "UNEXPECTED ERROR";ERRN
700            Branch=6
710         END SELECT
720      END IF
730   END IF
740   ON Branch GOTO 270,250,260,280,290,10
750   END
```

ON ERROR CALL

ON ERROR CALL is global, meaning once it is activated, the specified subprogram will be called immediately whenever an error is encountered regardless of the current context. System priority is set to level 16 inside the subprogram, and remains that way until the SUBEXIT is executed, at which time the system priority will be restored to the value it was when the error happened.

The ON ERROR CALL statement should only be used when you can guarantee that the problem causing the error can be fixed and the line can be re-executed safely. Remember that if the action taken in the error service routine is not sufficient to correct the problem, the program will dive into an infinite loop. Every time an error occurs, a CALL will cause a branch to the error service routine which will return execution to the line causing the error when a SUBEXIT statement is executed.

Bear in mind that an ON...CALL statement cannot pass parameters to the specified subprogram, so the only way to communicate between the environment in which the error is declared and the error service routine is through a COM block.

The ERRL function will not work in a different environment than the one in which the ON ERROR statement is declared, so when using an ON ERROR CALL, you should set things up in such a manner that the line number either doesn't matter, or can be guaranteed to always be the same one when the error occurs. This can be accomplished by declaring the ON ERROR immediately before the line in question, and immediately using OFF ERROR after it.

```
5010   ON ERROR CALL Fix_disc
5020   ASSIGN @File TO "Data_file"
5030   OFF ERROR
5040   !
5050   !
5060   !
7020   SUB Fix_disc
7030   SELECT ERRN
7040   CASE 80
7050      DISP "Door open -- shut it and press CONT"
7060      PAUSE
7080   CASE 83
7090      DISP "Write protected -- fix and press CONT"
7100      PAUSE
7120   CASE 85
7130      DISP "Disc not initialized -- fix and press CONT"
7140      PAUSE
7160   CASE 56
7170      DISP "Creating Data_file"
7180      CREATE BDAT "Data_file",20
7190   CASE ELSE
7200      DISP "Unexpected error ";ERRN
7210      PAUSE
7220   SUBEND
```

ON ERROR RECOVER

The ON ERROR RECOVER statement sets up an immediate branch to the specified line whenever an error occurs. The line specified must be in the context of the ON...RECOVER statement. ON ERROR RECOVER is global in scope — it is active not only in the environment in which it is defined, but also in any subprograms called by the segment in which it is defined.

If an error is encountered while an ON ERROR RECOVER statement is active, the system will restore the context of the program segment which actually set up the branch, including its system priority, and will resume execution at the given line.

```
.
.
.
3250  ON ERROR RECOVER Give_up
3260  CALL Model_universe
3270  DISP "Successfully completed"
3280  STOP
3290 Give_up:  DISP "Failure ";ERRN
3300  END
.
.
.
```

Chapter 12 Program Debugging

Introduction

The problem of debugging a program is distinct from the issues raised in Chapter 11, Error Handling. Chapter 11 is based on the premise that the programmer is satisfied that the program works as it should, and that it then should be made as foolproof as possible. This could be construed as putting the cart before the horse — before you can make a program foolproof, you must get it to run correctly in the first place. One of the key characteristics of a "bug" is that it doesn't necessarily have to cause an error condition to occur — it only has to cause your program to give a wrong answer. This chapter deals with the methods available on this computer to diagnose problems in logic and semantics.

Naturally, the ideal way to debug a program is to write it correctly the first time through, and all programmers should strive constantly to achieve this state of nirvana. Hopefully, the techniques that have been been discussed in this manual will help you get a little closer to this goal. The practice of writing self documenting code and designing programs in a top down fashion should help immensely.

Aside from recommended methods of writing software, the computer itself has several features which aid in the process of debugging.

Using Live Keyboard

One of the pleasing characteristics of this computer is that its keyboard is "live" even during program execution. That is, you can issue commands to the computer while it is running a program the same way that you issue commands to it while it is idle. For instance, you can add two numbers together, examine the catalogue of the disc currently installed in the drive, list the running program to a printer, scroll the CRT alpha buffer up and down, enter and exit either the graphics or alpha displays, or output a command to a function generator over HP-IB. Practically the only thing you can't do from live keyboard while a program is running is write or modify program lines, or attempt to alter the control structures of the program. (A complete list of illegal keyboard operations is given a little later on.)

By way of illustration, key in the following program, press [RUN], and then execute the commands shown underneath the listing.

```
10    FOR I=1 TO 1.E+5
20    NEXT I
30    END
```

```
CAT
2+2
SQR(6^2+17.2^2)
PRINT "THE QUICK BROWN FOX"
TIMEDATE
```

Now, this program will take a fair amount of time to complete (about 18 seconds), so to find out how far the program has gone, merely type I and press [EXECUTE] or [RETURN]. The current value of I will be displayed at the bottom of the screen. Now if you don't want to wait for the program to go through all one hundred thousand iterations, you can merely change the value of I by executing the command

```
I=99999
```

Thus, we have seen that live keyboard can be used to examine and/or change the contents of the program's variables.

One aspect of live keyboard to be aware of is that the computer will only recognize variables that exist in the current program environment. For instance, suppose that we change our example program to call a subprogram inside the loop.

```
10    FOR I=1 TO 1.E+5
15      CALL Dummy
20    NEXT I
30    END
40    SUB Dummy
50    FOR J=1 TO 10
60    NEXT J
70    SUBEND
```

While this program is running and you test the variable I from the keyboard, chances are that you will only get a message saying that I doesn't exist in the current context — most of the time will be spent in the subprogram. On the other hand, if you test the value of J, it is highly likely that you will get an answer.

Similarly, operations like ASSIGN and ALLOCATE, which are declarative types of statements, must use variables that are already known to the current environment when they are executed from the keyboard. For example, in the following program, it is perfectly legal to perform the operation `ASSIGN @Dvm TO *` from the keyboard, although it is not legal to perform `ASSIGN @File TO "DATA"` from the keyboard.

```
1     ASSIGN @Dvm TO 724
10    FOR I=1 TO 1.E+5
20    NEXT I
30    END
```

Live keyboard operations are allowed to use variables already known by the running program. Live keyboard operations are not allowed to create variables.

Although the GOTO and GOSUB commands are illegal from the keyboard, it is perfectly legal to call subprograms from the keyboard. The only restriction on using SUB and function subprograms from the keyboard is that the parameters that are passed must either be constants or must be variables that exist in the current context.

Here is an example:

```
10    FOR I=1 TO 1.E+5
20    NEXT I
30    END
31    !
40    SUB Gather(INTEGER X)
50    OPTION BASE 1
60    DIM A(32)
70    CREATE BDAT "File"&VAL$(X),1
80    ASSIGN @Dvm TO 724
90    ASSIGN @File TO "File"&VAL$(X)
100   OUTPUT @Dvm;"N100S"
110   ENTER @Dvm;A(*)
120   OUTPUT @File;A(*)
130   PRINT A(*),
140   SUBEND
141   !
150   DEF FNPoly(X)
160   RETURN X^3+3*X^2+3*X+X
170   FNEND
```

By executing CALL Gather(1) from the keyboard, the main program will be suspended while the subprogram is called, at which time a 1 record file will be opened, 32 readings will be taken from the voltmeter and stored in the file, and the readings will be printed on the screen. Then main program execution will resume where it left off.

Similarly, by typing FNPoly(1), the value of the polynomial will be computed for X=1 and the answer (8) will be displayed at the bottom of the screen.

Here is a list of commands which may not be executed from the keyboard while a program is running, although they may be executed from the keyboard if the computer is idle:

RUN	SCRATCH	GET
CONT	SCRATCH A	LOAD
EDIT	SCRATCH C	LOAD BIN
DEL	SCRATCH BIN	SYSBOOT

Stepping

One of the most powerful debugging tools available is the capability of single stepping a program, one line at a time. This process allows the programmer to examine the values of his variables and the sequence in which the program is running at each statement. This is done with the [STEP] key.

There are three ways to use the [STEP] key:

1. If the program is stopped (i.e., a prerun has to be performed), pressing the [STEP] key will cause the system to perform a prerun on the program, but no program lines will actually be executed. The first line that will be executed will appear in the system message line at the bottom of the screen. Pressing the [STEP] key again will cause that line to be executed, and the next line after that to be executed will appear in the message line. If the [STEP] key is pressed causing the next line to appear in the display, and a live keyboard operation (such as examining the value of a variable) is performed, the contents of the message line will change. Pressing the [STEP] key again will still cause the line to be executed, even though it is no longer visible in the implied display line. After the statement has completed, the next line will again appear.
2. If the program is in an INPUT or LINPUT statement, pressing the [STEP] key is sufficient to terminate the operation. Any data entered from the keyboard will be entered into the correct variables, just as though [CONTINUE] or [ENTER] had been pressed, but program execution will be PAUSEd, and the statement immediately following the INPUT or LINPUT will appear in the system message line.
3. If the program is in a PAUSEd state, pressing the [STEP] key will cause the next line to be executed. The program counter will not be reset, nor will a prerun be performed. Again, the next line to be executed will appear in the system message line after the last one has been completed. A paused state is indicated by a dash in the run light in the lower right hand corner of the screen.

Type in the following example and execute it by pressing the [STEP] key repeatedly.

```
10  DIM A(1:5)
20  ! This is an example
30  S=0
40  FOR I=1 TO 5
50  INPUT "Enter a number",A(I)
60  S=S+A(I)
70  NEXT I
80  PRINT S
90  PRINT A(*);
100 END
```

Notice that the [STEP] key caused every statement to appear in the system message line, one at a time, even those statements that are not really executed, like DIM and comments.

Tracing

The process of single stepping, wonderful though it is, can be quite slow, especially if the programmer has little or no idea which part of his program is causing the bug. An alternative way of examining variable changes and program flow is available in the form of the TRACE ALL statement.

TRACE ALL

When the TRACE ALL command is executed, it causes the system to issue a message prior to executing every line (this shows the order in which the statements were executed), and if the statement caused any variables to change value, a message telling the variables involved and their new values is also issued. The messages are issued to the system message line, and the most useful way to use the TRACE ALL feature is to turn Print All On (use the PRT ALL key), unless of course you're a very fast reader. (The printall mode will cause all information from the DISP line, the keyboard input line, and the system message line to be logged on the PRINTALL IS device.)

Turn Print All ON and key in the following example to see how TRACE ALL works:

```
10   TRACE ALL
20   FOR I=1 TO 10
30      PRINT I;
40      IF I MOD 2 THEN
50            PRINT " is odd."
60      ELSE
70            PRINT " is even."
80      END IF
90   NEXT I
100  END
```

There are two optional parameters that can be used with TRACE ALL. Both parameters are line identifiers (line numbers or line labels). The first parameter tells the system when to start tracing, and the second one (if it's specified) tells the system when to stop tracing. The following example illustrates the use of one optional line specifier:

```
1    TRACE ALL 40
10   DIM A(1:10)
20   FOR I=1 TO 100
30   NEXT I
40   FOR J=1 TO 10
50   A(J)=J
60   NEXT J
70   END
```

It is usually more useful to use the TRACE ALL command from the keyboard rather than from the program because a program modification is not necessary if you want to trace a different part of the program. All that's necessary is to type in a new TRACE ALL command from the keyboard to override the old one. In the above example, to trace the loop from 20 to 30 instead of the one from 40 to 60, simply delete line 1 and type in `TRACE ALL 20,40` from the keyboard.

```
10   DIM A(1:10)
20   FOR I=1 TO 100
30   NEXT I
40   FOR J=1 TO 10
50   A(J)=J
60   NEXT J
70   END
```

The program will begin tracing at line 20, and keep on tracing until it's ready to execute line 40, at which time it will terminate the trace messages and will continue executing the program normally.

If the TRACE ALL statement uses a line label instead of a line number, be aware of what happens if you have more than one occurence of a given line label in your program. For instance, it is perfectly legal to have the same line label in two or more different program environments — line labels are local to subprograms and branching operations addressing a given line label are treated separately in different subprograms. However, when a TRACE ALL using a line label is executed, the first line label in memory is the one that gets used, regardless of the environment the progam was in when the TRACE ALL statement was executed. Thus in the following program, even though the TRACE ALL Printout statement is executed inside the subprogram, tracing does not commence until the subprogram has been exited and the Printout statement in the main program has been executed.

```
10   DIM A(1:10)
20   FOR I=1 TO 10
30      CALL Dummy(A(*),I)
40      GOSUB Printout
50   NEXT I
60   STOP
70 Printout: !
80   FOR J=1 TO 10
90   PRINT A(J);",";
100  NEXT J
105  PRINT
110  RETURN
120  END
130  SUB Dummy(X(*),Z)
140  TRACE ALL Printout
150  FOR I=1 TO 10
160     X(I)=Z*100+I
170  NEXT I
180  GOSUB Printout
190  SUBEXIT
200 Printout: !
210  PRINT "Dummy routine executed";Z
220  RETURN
230  SUBEND
```

If two line identifiers are used, their location with respect to each other does not matter. Tracing will start when the line specified first is encountered, and it will stop when (or if) the second line is encountered.

PRINTALL IS

The PRINTALL IS command is useful for switching the tracing messages between the CRT and a hardcopy printer. (Again, to get any record at all of the trace messages, Print All must be On.) To cause the trace messages to be logged on the CRT, execute PRINTALL IS CRT. (The CRT is the default PRINTALL IS device that the system assumes when it wakes up.) To cause the messages to be logged on a printer, merely change the device selector to the appropriate value (PRINTALL IS 701).

TRACE PAUSE

The TRACE PAUSE command can be used to set a "break point" in the program. The program will execute at a reduced speed until the specified line is reached, at which time the program will pause, and the specified line will be shown in the implied display line, indicating that the program will execute it when execution is resumed. Execution may be resumed with the CONTINUE key, the STEP key (which will only cause one line to be executed), or by executing CONT from the keyboard (the specified line identifier must be located in the current environment).

By executing the command TRACE PAUSE Printout from the keyboard, the following program will pause every time it reaches line 70.

```
10    DIM A(1:10)
20    FOR I=1 TO 10
40       GOSUB Printout
50    NEXT I
60    STOP
70 Printout: !
80    FOR J=1 TO 10
90    PRINT A(J);",";
100   NEXT J
110   PRINT
120   RETURN
130   END
```

Try the following ways of continuing execution:

press STEP
press CONTINUE
execute `CONT 110`

As with TRACE ALL, a new TRACE PAUSE statement overrides a previous one. The same rules are applied when a line label is used in a TRACE PAUSE statement as are applied to the TRACE ALL statement — the first line in memory having that label is used.

TRACE OFF

TRACE OFF cancels the effects of any active TRACE ALL or TRACE PAUSE statements. The status of Print All and the PRINTALL IS device will be unchanged.

TRACE OFF may be executed either from the program, or from the keyboard.

The CLR I/O Key

The CLR I/O key (BREAK on HP 46020A keyboards) suspends any active I/O operation and pauses the program in such a way that the suspended statement will restart once CONTINUE or STEP is pressed. This is useful for operations which appear to "hang" the machine, such as printing to a printer which isn't turned on.

Most devices will not respond to ENTER requests unless they have first been instructed to respond. If improper values are sent to a device, it may refuse to respond. Therefore, CLR I/O can help in debugging these situations.

Here are the operations that can be suspended with CLR I/O.

PRINT	SEND	ASSIGN
LIST	PRINTALL outputs	PURGE
CAT	ENTER	CREATE
OUTPUT	INPUT	Some graphics commands
DUMP GRAPHICS	HP-IB commands	
DUMP ALPHA	External plotter commands	

Efficient Use of the Computer's Resources

Chapter 13

Introduction

Every model of computer has certain characteristics which can result in better performance, provided the programmer knows what those characteristics are and how he can take advantage of them. This chapter consists of a potpourri of such items.

Data Storage

Data Storage in Read/Write Memory

There are four data types on this computer: REAL, INTEGER, strings, and I/O path names. The R/W memory occupied by data is made up of two parts: the memory it actually takes to hold the intended information, and the memory that the system uses to keep track of the information's location and form (this is called overhead). Strings, INTEGERs, and REALs can be declared either as simple variables or as arrays. Arrays take different amounts of overhead than simple variables, but each element of an array uses the same amount of memory that a corresponding simple variable uses to actually store information.

The overhead required for any given symbol is kept in three tables: the symbol table, the token table and the dimension table. The symbol table contains pointers to the value area, where the actual information is kept, and to the other two tables. The token table contains the names of the various symbols. The dimension table contains length information for strings and arrays, and is not used for numeric scalers. The tables are not constructed in single units as symbols are added and deleted. Rather, as new space is required, the system will first look to see if there are any unused entries in the tables — if new space is allocated, usually enough for several entries is allocated. For instance, the symbol table is built in increments of five entries.

Symbol Table Overhead:	10 bytes per symbol
Token Table Overhead:	number of characters in the name + 1 (if the above number is odd, it is rounded up to an even number). Note that the name for I/O path names, strings, and functions includes the @, $, and FN, respectively.
Dimension Table Overhead:	For arrays: 3 bytes (total size) 1 byte (number of dimensions) 4 bytes for each dimension (for the lower bound, and the size of each dimension) For strings: 2 bytes (maximum length) For string arrays — all of the normal array overhead, plus two bytes for the maximum allowed length of an element

Note that line labels, COM labels, and subprograms are considered as symbols, and occupy space in both the symbol and token tables. Line numbers used **in** statements, like GOTO 20, also occupy space in the symbol table.

Every subprogram (or context) has its own set of tables. In addition, there is a global set of COM tables, where all information concerning COM blocks is kept. Symbols that belong to a COM block will occur in both the COM tables and in any local tables in which that COM block is declared. Since each context may define the names by which it refers to COM block variables, there will be no entry in the COM token table for each variable, but an entry in the COM token table **will** occur for COM labels.

ALLOCATEd variables require four bytes of overhead in addition to the overhead already discussed for the symbol, token, and dimension tables.

The following table summarizes the storage requirements for various data types. This table does not show the extra requirements just mentioned for ALLOCATEd and COM variables.

Type	Overhead	Information Storage
Simple INTEGER	10 bytes + name overhead	2 bytes
Simple REAL	10 bytes + name overhead	8 bytes
Simple string	12 bytes + name overhead	1 byte per char. up to declared length (padded to even number of chars.) + 2 bytes (length information)
I/O path name	10 bytes + name overhead	100 bytes
INTEGER array	14 bytes + name overhead + 4 bytes per dimension	2 bytes per element
REAL array	14 bytes + name overhead + 4 bytes per dimension	8 bytes per element
String array	16 bytes + name overhead + 4 bytes per dimension	1 byte per char. up to declared length (padded to even number of chars.) + 2 bytes (length information) per element

Data Storage on Mass Memory Devices

The amount of storage that data takes on mass storage media is similar to the amount of R/W memory that data takes internally, except that no overhead is required (on BDAT files). Arrays and single values are interchangeable on mass storage — no distinguishing information is kept on the media.

INTEGERs (and INTEGER arrays)	2 bytes (per element)
REALs (and REAL arrays)	8 bytes (per element)
Strings (and string arrays)	4 bytes + 1 byte per char up to current length, padded to even number of chars. (per element)

For ASCII files, all information is converted to string (or ASCII) form, and a two-byte length field is tacked onto the front of every field.

INTEGERs (and INTEGER arrays)	2 bytes + 1 byte per digit (per element)
REALs (and REAL arrays)	2 bytes + 1 byte per digit (per element)
Strings (and string arrays)	2 bytes + 1 byte per char (per element)

Comments and Multicharacter Identifiers

Self-documenting features such as in-line comments and multicharacter variables and line labels are useful because of the benefits to be reaped in terms of developing, testing, debugging, and maintaining programs. They do take extra memory, but this shouldn't be a problem if you keep the following points in mind.

Comments take 1 byte of R/W memory for every character in the comment. If memory space becomes a problem, many people resort to keeping two copies of their programs around — one fully commented to use as reference material, and the other uncommented to use as the "production version", which is the one that is actually used.

Multicharacter identifiers are only spelled out in their entirety once — not every time they are used. The program actually stores pointers whenever a reference to the identifier is used, so using short identifiers won't result in any appreciable savings in memory used.

Variable and Array Initialization

Care should be taken to initialize any variables before using them in an expression (on the right hand side of an =, as a left-hand subscript in a function or subprogram parameter list, as an argument to a built-in function, or in a PRINT/OUTPUT/DISP list). The system will set variables to zero, strings to null, and I/O path names to undefined at program prerun, but depending upon defaults like this is considered bad programming practice and could lead to subtle errors. For instance, the first time a certain line is executed, the variables used may be assumed to be zero because of the prerun operations. Once this assumption has been made, the danger is that the programmer will branch back to the same section of code and forget that the zeroing process has not been performed — an error may result that didn't occur previously.

Mass Memory Performance

Program Files

There are two ways to store programs — they can be saved either as ASCII source strings using the SAVE command, or they can be stored in an intermediate form that the BASIC language system understands using the STORE command.

If the time it takes to load the program is important, always use the STORE command to store the program instead of the SAVE command. The LOAD command, which reads in files created by the STORE command, will execute about fifty times faster than the GET command. This is because the LOAD command does not require that the information on the file be processed in any way. Since the program is already in the form the system needs it in, all that is necessary is to funnel the program directly into memory as fast as the disc can spin (assuming an interleave of one).

SAVE files, on the other hand, require that the system parse and check the lines as they are read, just the same as if a user had typed them in from the keyboard. Consequently, the speed at which the program gets loaded into memory with the GET command will be drastically slower than the LOAD command. Using the Models 226 and 236 internal drives as an example of the relative speeds, a typical 8K byte program will take about 30 seconds to GET, but only about one second to LOAD.

One advantage of the GET/SAVE commands is that it is possible to deal with programs as string data.

Data Files

As with program files, there are two types of data files: ASCII and BDAT. ASCII files require that all data be in string form, while BDAT files are interpreted as internal data representations.

When reading or writing data to an ASCII file, the number formatter is required to convert the data in between its internal representation and its ASCII form. When reading or writing data to a BDAT file the data may stream directly back and forth with no conversion required. Using the Models 226 and 236 internal drives as an example, an 8K element REAL array (64K bytes) may take around 200 seconds to write in an ASCII file, while the same array will only take about 5 seconds to write to a BDAT file.

The primary benefit of the ASCII data file is the transportation of data between different models of Hewlett-Packard computers and terminals and between discs used with different language systems.

Benchmarking Techniques

This section discusses the techniques used to determine how fast various operations execute. Ideally, you should separate the measurement time from elapsed time:

```
10   T1=TIMEDATE
20   T2=TIMEDATE
30   PRINT T1-T2;"seconds used to read clock"
40   END
```

In actuality, the clock only has a resolution of 10 ms, so you won't usually be able to time this operation.

Next, most operations are performed inside a loop in order to be able to time operations that are faster than the resolution of the clock (clock resolution is 10 ms.). This also tends to "smooth out" varying system overhead characteristics.

```
10   INTEGER I
20   T1=TIMEDATE
30   FOR I=1 TO 10000
40   NEXT I
50   T2=TIMEDATE
60   PRINT T2-T1;"seconds of loop overhead"
70   END
```

A certain amount of time used in computational operations will involve moving information around. The time will be different depending upon the type of the information being moved (string, REAL, or INTEGER), and for strings, the length.

```
10   REAL A,B,C
20   INTEGER I
30   B=PI
40   T1=TIMEDATE
50   FOR I=1 TO 10000
60   A=B
70   NEXT I
80   T2=TIMEDATE
90   PRINT T2-T1;"seconds of loop overhead"
100  END
```

The next step is to actually time the operation of interest. It should be noted that for arithmetic operations, the time spent performing the operation will vary depending upon the two operands (number of digits and relative magnitudes).

```
10   REAL A,B,C
20   INTEGER I
30   B=PI*1.E+53
40   C=EXP(SQR(2)^13.81)
50   PRINT "B=";B,"C=";C
60   T1=TIMEDATE
70   FOR I=1 TO 10000
80     A=B
90   NEXT I
100  T2=TIMEDATE
110  FOR I=1 TO 10000
120    A=B+C
130  NEXT I
140  T3=TIMEDATE
150  Op_time=DROUND(T3-T2-T2+T1,3)
160  PRINT Op_time*100;"us. per operation"
170  END
```

The above program will show anywhere from 148 to 150 microseconds per operation for addition.

Here is a list of a few other operations:

Addition	150 μs
Subtraction	165 μs
Multiplication	301 μs
Division	460 μs
Exponentiation	7590 μs

These times vary for different processor boards. Use these times and others throughout this chapter to compare the speeds of different operations.

INTEGER Variables

We have seen in the first section of this chapter that INTEGER variables don't take as much memory as REAL variables (2 bytes instead of 8). Now we shall discover that some operations with INTEGERs are much faster than the same operations with REALs.

Minimum and Maximum Values

The INTEGER variable type may store any whole number from −32 768 to +32 767 inclusive.

Mathematical Operations

There are two sets of math routines provided for the MOD, DIV, +, −, and * operations: REAL and INTEGER. Depending upon the types of the operands used, the execution times for these operations will vary widely. The tradeoffs are:

INTEGER math is the faster of the two, since it doesn't require as much "work". This is because:

1. There are only two bytes of data to process instead of eight
2. Operations do not have to deal with a combination of mantissa and exponent.
3. The results don't have to be normalized.
4. INTEGER math can be done directly in the hardware.

REAL math, though slower, is generally more widely used because it allows numbers with fractional parts to be analyzed. REAL numbers carry about 16 decimal digits of precision and have an exponent range of −308 to +308.

Note

All times specified are without the floating point card. If you have this card, your times will be faster for REAL math.

For instance, suppose you want to compute your monthly pay. Assume that you're making $5.17 an hour, that you work twenty four days per month and that you work 14 hours per day. The calculation that you would use is `5.17*24*14` or $1737.12. In this problem, you definitely want your computer to use REAL precision math (or you'll lose 17 cents per hour!) even though you're only using 6 of the 16 digits available.

The computer will pick whatever math routines it needs to solve the current problem. However, the programmer can exercise control over which math routines get executed if the following rules are understood.

- INTEGER math is used if both arguments of a MOD, DIV, *, +, or −operation are of type INTEGER. If the results of the operation cannot be stored in an INTEGER, then an error is generated (INTEGER overflow).
- REAL math is used if either or both arguments of a MOD, DIV, *, +, or − operation is of type REAL. If one of the arguments is of type INTEGER, then that argument is first converted to REAL.
- REAL math is always used for exponentiation and division (slash).

The following table gives some approximate time comparisons between INTEGER and REAL operations for +, −, and *. The times are approximations because REAL math routines do different things depending upon the values of the operands. All times shown here were found on operations with numbers having no fractional parts. The multiplication times were found for operands in the range of −200 to +200.

	REAL	INTEGER
MOD	160 µs	91 µs
DIV	352 µs	88 µs
Addition	142 µs	68 µs
Subtraction	174 µs	68 µs
Multiplication	152 µs	77 .113s

Multiplication, like most math operations, will execute faster on INTEGER values. However, bear in mind that it's much easier to get an INTEGER overflow on multiplications than on additions and subtractions. For instance, `200*200` will give an INTEGER overflow. If you are performing multiplication on INTEGERs, you should carefully examine your program to ensure that the range of your answers doesn't force you to use REALs, even if the requirement for fractional precision doesn't.

Loops

In general, any FOR/NEXT loop using an index which has been declared to be an INTEGER will execute about 2.4 times faster than a loop whose loop counter is a REAL. Type in the two programs below and run them to see the difference.

```
10   REAL I
20   TO=TIMEDATE
30   FOR I=1 TO 10000
40   NEXT I
50   PRINT TIMEDATE-TO;"seconds"
60   END
```

Time is about 1.67 seconds.

```
10   INTEGER I
20   TO=TIMEDATE
30   FOR I=1 TO 10000
40   NEXT I
50   PRINT TIMEDATE-TO;"seconds"
60   END
```

Time is about .69 seconds.

Bear in mind that the 2.4 speed improvement is only on the time devoted to actually incrementing and testing the loop variable (in these examples, I). So, any loop that iterates for 10 000 times will run about a second faster if the index is an INTEGER instead of a REAL. Now, saving a second on a loop that executes 10 000 times may not sound like much by itself, and it's not. But what if that loop is nested inside **another** one that executes 10 000 times? Now your time savings is 10 000 seconds, or two hours and forty-five minutes! Just for declaring the loop counters to be INTEGER.

Naturally, making a loop index into an INTEGER will only work if the loop is not stepping in fractions, and if the minimum and maximum values of the loop index do not exceed the range of −32 768 thru +32 767.

Array Indexing

Accessing individual array elements is faster if the variables or expressions giving the indices into the array are INTEGER instead of REAL. This is because the system has to convert floating point numbers into an INTEGER in order to find the offset from the beginning of the array. If the index is already in INTEGER form, the conversion isn't necessary. The following example illustrates this point.

```
10   REAL I
20   DIM A(1:1000)
30   X=17.568
40   T0=TIMEDATE
50   FOR I=1 TO 1000
60      A(I)=X
70   NEXT I
80   PRINT TIMEDATE-T0;"seconds"
90   END
```

```
10   INTEGER I
20   DIM A(1:1000)
30   X=17.568
40   T0=TIMEDATE
50   FOR I=1 TO 1000
60      A(I)=X
70   NEXT I
80   PRINT TIMEDATE-T0;"seconds"
90   END
```

You will find a difference of .14 seconds between the two programs' execution times, due to a combination of the loop counter being INTEGER and the INTEGER indexing of the array. Again, if you're operating on a much larger array, or if you're working on a multi-dimensional array this number can become noticeable.

REAL Numbers

Minimum and Maximum Values

The minimum REAL number that can be stored on this computer is approximately $\pm 2.225\,073\,858\,507\,202 \times 10^{-308}$

The maximum REAL number that can be stored on this computer is approximately $\pm 1.797\,693\,134\,862\,315 \times 10^{308}$

A REAL number can also have the value zero.

Type Conversions

Earlier, it was mentioned that any time a MOD, DIV, *, +, or −operation is performed on two numbers of different type (one INTEGER, and one REAL), a type conversion has to take place to convert the INTEGER to a REAL. This section will address other situations where type conversions have to take place.

Any time an INTEGER is used in an exponentiation or division operation, it must first be converted to a REAL.

All of the following functions require a REAL argument (with the exception of VAL and RND), and all of them return a REAL value (with the exception of RANDOMIZE). If an INTEGER is passed in, or if the result is to be stored in an INTEGER, then the appropriate type conversion must be made: EXP, LGT, LOG, RANDOMIZE, SQR, DROUND, RND, ACS, COS, ASN, SIN, ATN, TAN, VAL.

All of the comparison operators (=, <>, <, >, <=, >=) will return INTEGER values (0 or 1) but will accept either INTEGERs or REALs as arguments. The logical operators AND, EXOR, OR, and NOT will convert any arguments to the INTEGER values 0 or 1 before the operation is performed, and an INTEGER 0 or 1 will be returned.

The binary bit functions (BINAND, SHIFT, ROTATE, BINIOR, BINCMP, BIT, BINEOR) require INTEGER inputs and provide INTEGER outputs. Type conversions will be performed if REALs are supplied as inputs, or if the results are to be stored in a REAL variable.

SGN returns an INTEGER (−1, 0, 1) regardless of the type of the argument passed to it. ABS and INT return the type of the argument that's passed to them.

If two INTEGERs are used to perform a MOD, DIV, *, +, or −operation, but the result is to be stored in a REAL variable instead of an INTEGER, then the result must be converted from INTEGER to REAL.

Here is how long each type conversion takes:

INTEGER to REAL: 42 microseconds
REAL to INTEGER: 34 microseconds

Constants

All constants that are within the range of −32 767 to 32 767 that aren't entered with a decimal point or an "E" (for scientific notation) are stored in the machine as INTEGERs. Integer constants should always be used with INTEGER variables, but if they are used with REAL variables they will have to be converted to REAL first. This operation will slow down the execution of the program, as shown in the previous section. Any numbers entered with decimal points (1.0, 3., .7, etc.), with an E (1E−304, .2E48, 0E0, etc.), or outside the valid INTEGER range (40000, −75986, etc.) will be stored as REAL constants.

Polynomial Evaluations

The polynomial can waste much of computer time because programmers tend to pick the most obvious, and also the most time-consuming, method of evaluating them. Polynomials are usually written mathematically as:

$$y = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

or

$$y = \sum_{i=0}^{n} a_i x^i$$

hence the temptation is strong to evaluate a polynomial on a computer as:

```
2000 DEF FNPoly(X,Coefficient(*),INTEGER N)
2010 INTEGER I
2020 Y=0
2030 FOR I=0 TO N
2040     Y=Y+Coefficient(I)*(X^I)
2050 NEXT I
2060 RETURN Y
2070 FNEND
```

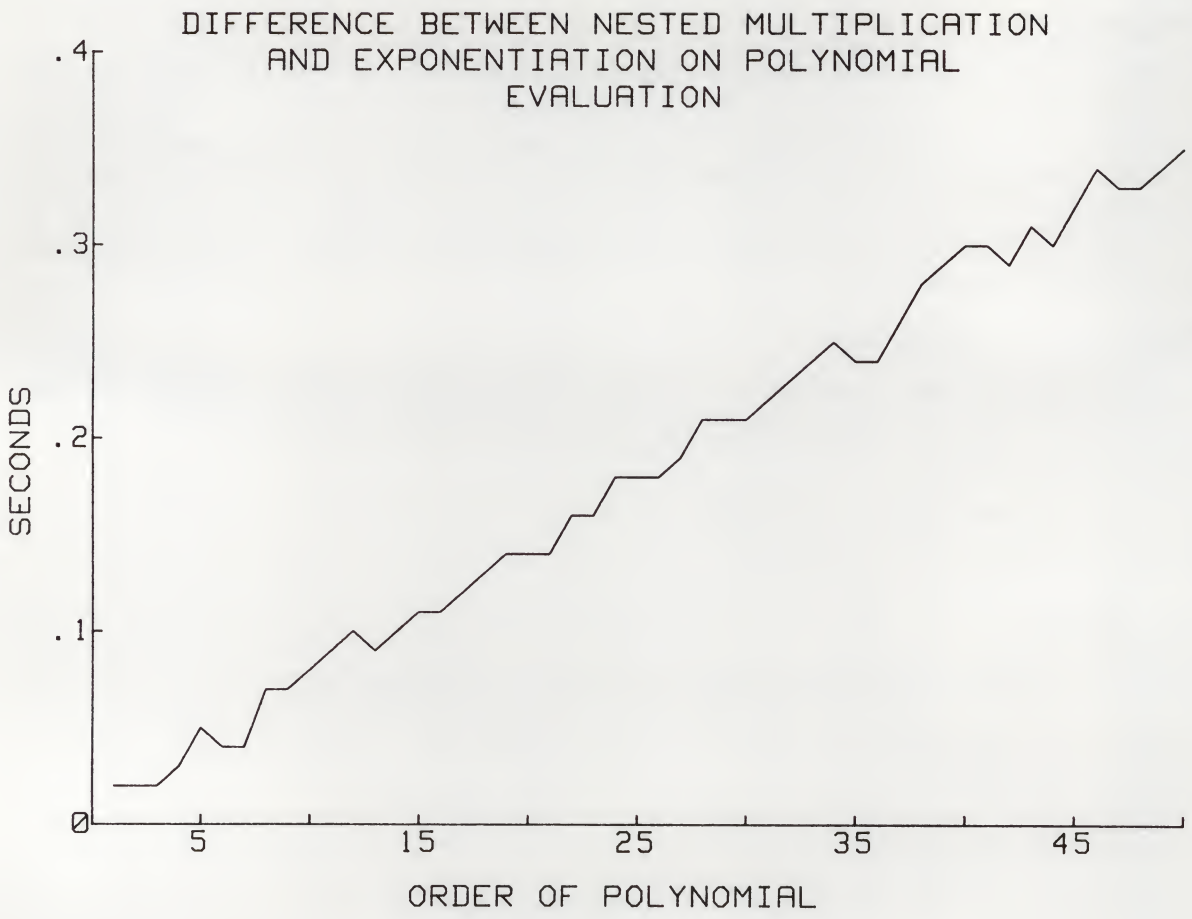
In the above program, there are N + 1 additions, N + 1 multiplies, N + 1 exponentiations, and N + 1 INTEGER to REAL conversions (I is converted to a REAL when the exponentiation operation is performed). Now suppose the polynomial is written in the equivalent form:

$$y = a_0 + x(a_1 + x(a_2 + \ldots + x(a_n) \ldots))$$

Then the corresponding program would be:

```
2000 DEF FNPoly(X,Coefficient(*),INTEGER N)
2010 INTEGER I
2020 Y=Coefficient(N)
2030 FOR I=N-1 TO 0 STEP -1
2040     Y=Coefficient(I)+X*Y
2050 NEXT I
2060 RETURN Y
2070 FNEND
```

Now there are only N additions and N multiplies, and NO exponentiations or INTEGER to REAL conversions! The following chart shows the time savings as a function of the order of the polynomial. For example, if the polynomial is of order 10, it is 70 milliseconds faster to use the nested multiply method to evaluate the polynomial than to use exponentiation. If you're plotting a thousand points on a graph, nested multiplication will save you more than a minute!



Logical Comparisons for Equality on REAL Numbers

Don't do it.

If you are performing mathematical operations on REAL numbers, and then comparing them for equality, the chances are that they will never come up equal. For example, in the previous section on polynomial evaluation, you can pass the same value for X and the same coefficient array to each of the two functions, and although the results will look equal when you print them out, they won't show equality if you compare them. (Try it and see.) A shorter example is to type out this expression from the keyboard and press EXECUTE:

```
.1+.1+.1+.1+.1+.1+.1=.7
```

The 0 at the bottom of the screen means that the machine doesn't consider the two numbers to be equal. This is characteristic of the way that binary math works.

Any REAL numbers should be rounded first before being tested for equality. This is one of the purposes of the DROUND function.

```
DROUND(.1+.1+.1+.1+.1+.1+.1,12)=DROUND(.7,12)
```

After rounding both numbers to 12 digits, the computer will now accept them as being equal. See Chapter 4 for more discussion on the comparison of REAL numbers.

Saving Time

Multiply vs. Add

It is always faster to add a number to itself than it is to multiply it by 2. For instance, to perform 2*PI a thousand times takes .22 seconds, while to perform PI+PI a thousand times takes .13 seconds.

However, if you want to multiply by 3, that is faster than adding the number three times. 3*PI executed a thousand times takes about the same as 2*PI (.22 seconds), but adding PI+PI+PI a thousand times takes about .28 seconds.

Exponentiation vs. Multiply and SQR

Exponentiation is very slow when compared to other mathematical operations. The results are worth the wait when the exponent has a fractional part; raising a REAL number to a REAL power is a complex operation. However, time can be saved if you are alert to some special cases. The most common examples are squaring a number or finding a square root. Using `SQR(X)` takes only about one-fourth the time required by the expression `X^.5`. Even more dramatic savings can be gained when raising numbers to an integer power. Using `X*X` yields a 22-to-1 time savings over the expression `X^2`. When using powers greater than 2 or 3, the expressions needed to achieve the repeated multiplication can be somewhat cumbersome, and may not even fit on a program line. However, repeated multiplication is so much faster than exponentiation that time savings can be realized (for powers up to 14) even if a FOR...NEXT loop has to be added to repeat the multiplication.

Array Fetches vs. Simple Variables

It takes more time to access an array element than it does a simple variable. This is because the address of the array element has to be computed from the starting address of the array and the offset within the array based on the specified indices. A simple variable's address does not require this computation.

Thus, if you access a given array element often enough, especially within a loop, it will often be faster to store the array element into a simple variable and then operate on the simple variable.

Time to fetch a simple variable and store it:	136 μs
Time to fetch an array variable and store it:	260 μs
Difference:	124 μs

This means that it is faster to fetch three simple variables than it is to fetch two array elements.

Concatenation vs. Substring Placement

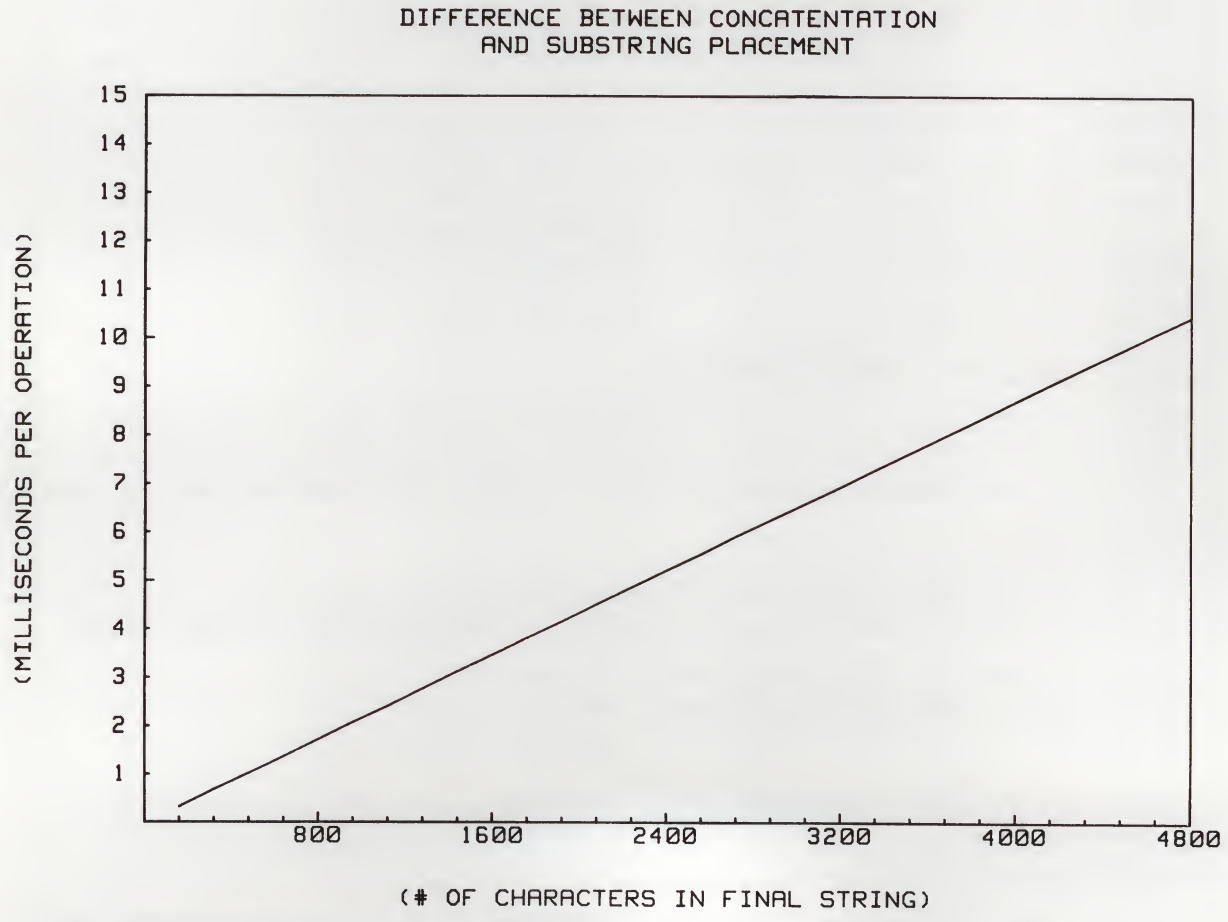
The concatenation operator (`&`) allows you to combine two or more strings to construct another string. This operation is discussed in Chapter 5. However, there is a special case that can be accomplished faster without the concatenation operator. This is the case where the new string is built by appending to the end of an existing string. For example, `A$=A$&B$`.

Concatenation works by constructing a temporary workspace in which all the components are assembled. Then the result is moved to its destination. In the example above, A$ is moved to a temporary workspace, B$ is appended to it, and the result is moved back to A$. Thus, the original contents of A$, which weren't changed, have been moved twice unnecessarily. A faster way to accomplish the same thing is:

```
A$[LEN(A$)+1]=B$
```

Another benefit of this approach is that the temporary workspace is not created. If memory is tight and A$ is very large, concatenation could create a memory overflow.

The following chart shows the time savings that result from using substring placement instead of concatenation.



Saving Memory

The ALLOCATE and DEALLOCATE statements can be used to make efficient use of memory space in certain applications. They are useful in programs that contain a number of large variables that are not all needed simultaneously. For example: during data collection, a large string array is needed; during data processing a large numeric look-up table is needed; and during output formatting, a string array is needed again. Because a mass storage device is used to hold the data between processes, the same memory area can be used for all these arrays.

To use ALLOCATE effectively, it is necessary to understand how the system reclaims areas that have been DEALLOCATED. Space for allocated variables is built using a stack discipline. The DEALLOCATE statement marks a space as unused. Unused space can be reclaimed only if it is the *last* space on the stack. There are two operations that use space in this stack. One is ALLOCATE, and the other is ON <event>.

Keeping other allocated variables from blocking deallocated space is relatively simple. If you have only one allocated variable at any given time, this is not a problem. If you have allocated variables in existence simultaneously, ALLOCATE them in the opposite order of the DEALLOCATE statements. Think of this in the same way you would think about nesting FOR...NEXT loops.

Preventing blockage by ON conditions is more complicated. ON conditions, with one exception, create control blocks that are *permanent* entries on the stack. As soon as you declare an ON (or OFF) condition, all the previous entries on the stack are "locked in" for the duration of the context and cannot be reclaimed. Therefore, all the control blocks should be created *before* any variables are allocated. Once a control block is created, it will be used by all subsequent ON and OFF statements that refer to the same resource. A good technique is to include an OFF statement for each desired event before allocating any variables.

The exception mentioned above is an ON condition declared for an I/O path name. This includes ON END, ON EOT, and ON EOR. For these, subsequent ON and OFF statements behave as previously described. However, if the I/O path is closed, any control blocks associated with the path are marked as unused. This has two implications. One, the reclaiming of the stack will not be blocked after the I/O path is closed. Two, you cannot force the system to leave these control blocks at the beginning of the stack. Here is an example:

```
200  ASSIGN @File to "FRED"
210  ON END @File GOTO Label1
220  ALLOCATE Array(255,4)
.
.
.
600  ASSIGN @File TO "SUSAN"
610  ON END @File GOTO Label2
620  DEALLOCATE Array(*)
```

At first, the array and control block are allocated in the proper order. The ASSIGN statement in line 600 closes the original path and opens a new path with the same name. When the ON END control block for the new path is created, it is placed after the array on the stack. Therefore, no memory space can be recovered by deallocating the array.

Notes

Chapter 14 Graphics

Introduction

Graphics is a powerful tool for presenting information. Computer graphics can be equally powerful but an extra step is required between the conception of the idea and the final image. This step is the construction of a mathematical model of the image within the computer.

Since computers only do what they are told, it is essential to have a complete knowledge of the commands that communicate between the real world and the computer's world. This knowledge is needed to create the model within the computer's memory and to understand the resulting image of the model.

This chapter introduces basic graphics techniques. For more extensive information and examples, see the *BASIC Graphics Techniques* manual.

A good place to start is the screen of the CRT. The screen serves a dual role; as an output device for text and as a plotting device for graphics. Although many applications require a plotter to produce the final image on paper, it is easier to develop the model on the screen. Then with minor changes the image can be sent to an external plotter.

A non-bit mapped display is capable of displaying two rasters (images) either individually or combined. The alpha raster displays alpha-numeric characters while the graphics raster displays pixels (picture elements). The arrangement for the Model 216 and Model 226 is 400 columns of 300 pixels; for the Model 236, 512 columns of 390 pixels. (Other monitors are available. The arrangement for all monitors is listed in the *BASIC Language Reference* manual in the GSTORE statement.)

Display of the rasters is controlled by program statements (commands when executed immediately) or by keys located in the upper right corner of the keyboard.

- Press (GRAPHICS) to turn on the graphics raster. A second press turns off the alpha raster.
- Press (ALPHA) to turn on the alpha raster. A second press turns off the graphics raster.

The rasters are also controlled by the BASIC statements:

```
GRAPHICS ON and GRAPHICS OFF
ALPHA ON and ALPHA OFF
```

Initially the alpha raster is on and the graphics raster is off. During execution of a graphics program the alpha raster may be turned off but execution of an input statement turns the alpha raster back on and leaves it on.

Bit mapped displays do not have a separate alpha and graphics raster. Both are combined and always on.

Graphics Fundamentals

The First Step

One statement initializes the system by setting default values. Several operations are actually performed by this statement.

Execute the command.

```
GINIT
```

This statement (or its equivalent series of statements) should precede any other graphic statements. The system is now ready but nothing will be displayed until it is told to turn on the graphics raster.

Either execute the command,

```
GRAPHICS ON
```

or press the [GRAPHICS] key once.

Whenever you wish to clear the graphics raster, execute GCLEAR. GINIT also clears graphics raster but not until the execution of a graphics statement that selects a default plotter.

If the alpha raster is not clear, press the [CLR SCR] key.

Pencil and Paper

Traditionally, a drawing requires pencil and paper. Here the paper is replaced by a display screen (CRT raster) and the pencil is replaced by the PEN statement.

There are three currently defined pens for the monochrome graphics raster.

`PEN 1` turns pixels on. (white pen)

`PEN -1` turns pixels off. (black pen)

`PEN 0` complements pixels. (magic pen)

When drawing with PEN 0, black pixels become white and white pixels become black. If you plan to send the image to an external plotter, do not expect a pen to erase a line. Check the plotter's manual for different pen definitions.

After selecting a pen, several statements, and their variations, control the motion of the pen on the drawing surface.

`DRAW X,Y` draws a line to point X,Y.

`MOVE X,Y` moves the pen to point X,Y.

`PLOT X,Y,P` moves or draws to point X,Y.

DRAW and MOVE do exactly what their names imply. PLOT will do both moves and draws with optional pen control.

Try the following.

```
GINIT
GRAPHICS ON
DRAW 60,50
```

A white line should appear from the lower left corner to the middle of the screen. Why the lower left corner? Executing the statement GINIT returns the pen to location 0,0. This is currently the lower left corner of the display.

Execute the following commands to see their effect.

```
GCLEAR
PEN 1
MOVE 0,50
DRAW 100,50
MOVE 50,50
PEN -1
DRAW 0,50
PEN 0
DRAW 100,50
```

A white line is drawn across the screen, the pen moved to the midpoint of the line, a black pen draws a line to the left endpoint, and the line is then complemented.

To show the size of the current drawing area, the next statements draw a line around the perimeter, similar to the frame around a painting.

```
GCLEAR
PEN 1
MOVE 0,0
DRAW 100*RATIO,0
DRAW 100*RATIO,100
DRAW 0,100
DRAW 0,0
```

There is a statement that duplicates the result of the previous example. Try the following.

```
GCLEAR
FRAME
```

The Standard View

The default scale for the Model 226 and Model 216 display is 133 horizontal units by 100 vertical units. For the Model 236 it is 131 horizontal by 100 vertical. For the Model 237 (HP 98781A display) it is 133 horizontal by 100 vertical. These units are called **Graphic Display Units**. The origin (location 0,0) is in the lower left corner of the graphics raster. The pen can be positioned anywhere, even outside the drawing area. You may wish to experiment some more to verify this.

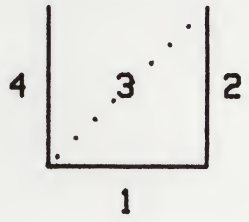
Pen Control

Pen action is automatically defined for the MOVE and DRAW statements. The pen is raised for a move and lowered for a draw. The PLOT statement has an optional pen control parameter that determines the pen's action according to the following table.

P value	Result
Odd	Pen down
Even	Pen up
Positive	Change pen after motion
Negative	Change pen before motion

The following example shows pen control by the optional pen control parameter.

```
GCLEAR
PLOT 20,20,1
PLOT 40,20
PLOT 40,40,0
PLOT 20,20,1
PLOT 20,40
```



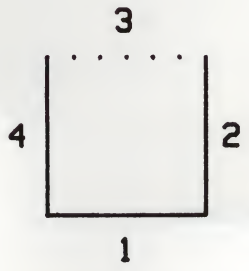
Incremental Pen Positioning

Often when plotting it does not matter where the pen has been, only the distance to the next desired pen position.

`IMOVE X,Y`	positive X moves pen to the right. negative X moves pen to the left.
`IDRAW X,Y`	positive Y moves pen up. negative Y moves pen down.
`IPLOT X,Y,P`	positive P changes pen after move. negative P changes pen before move.

Execute the following and watch the results.

```
GCLEAR
MOVE 50,50
IDRAW 10,0
IDRAW 0,10
IMOVE -10,0
IDRAW 0,-10
```



With each incremental motion of the pen, a new origin is created for any subsequent incremental pen statement. A third method of moving and drawing involves specifying the relative movements of the pen from a local origin.

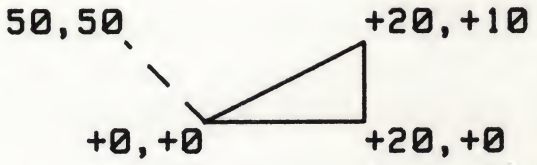
Relative Pen Positioning

The relative plot statement uses the current pen position as a local origin to define a second coordinate system. Pen motion is then relative to the local origin until the pen position is changed by a statement other than another RPLOT statement.

`RPLOT X,Y,P` where X and Y are relative displacemnts.

Execute these commands to draw a triangle.

```
GCLEAR
MOVE 50,50
RPLOT 20,10,-1
RPLOT 20,0
RPLOT 0,0
```



Note that the command RPLOT 0,0 returns the pen to the local origin (50,50) not the real origin (0,0). Since RPLOT creates a local origin it is useful when drawing a figure that needs to be repeated at different locations on the display.

Line Rotation

Lines generated by move, draw, and plot can be rotated by the PIVOT statement. The current angle mode is used to specify the rotation.

```
DEG
GCLEAR
PIVOT 45
MOVE 0,0
DRAW 100,0
```

`PIVOT 0` will return the system to normal.

How Colored Lines Are Drawn

Color displays compatible with the HP 98627 work on the principle of using combinations of three primary colors to represent the available colors. The primary colors are Red, Green, and Blue. Secondary colors are Yellow (Red combined with Green), Cyan (Green combined with Blue), and Magenta (Red combined with Blue). White is produced by combining the three primary colors, and Black is the lack of all three colors.

The primary colors may be conceptually grouped into three "planes" of dots, which are superimposed to create the visual color image. Thus, each pixel on the screen is composed of one primary-color dot from each of the three primary-color planes; the different colors are produced by turning on different combinations of dots.

When a line is drawn using one of the primary colors, dots in the primary-color plane are turned on by setting the appropriate bit in the corresponding primary-color memory plane. When a line is drawn using one of the secondary colors, two dots, one dot from each of two color planes, are turned on for each point drawn. For example, when Yellow is drawn, Red and Green dots are turned on at each point of the line to produce this secondary color. When White is drawn, all three primary colors are turned on at each point plotted. Drawing with the Complement pen may be more easily understood by studying the following chart and accompanying text.

Pen Number	Resultant Action	Effect on Each Color Plane:		
		Red	Green	Blue
−7	Erase Magenta	0	—	0
−6	Erase Blue	—	—	0
−5	Erase Cyan	—	0	0
−4	Erase Green	—	0	—
−3	Erase Yellow	0	0	—
−2	Erase Red	0	—	—
−1	Erase White	0	0	0
0	Complement	Invert Bit in Each Plane		
1	Draw White	1	1	1
2	Draw Red	1	0	0
3	Draw Yellow	1	1	0
4	Draw Green	0	1	0
5	Draw Cyan	0	1	1
6	Draw Blue	0	0	1
7	Draw Magenta	1	0	1

The chart shows how bits of each primary-color memory plane are affected by drawing lines with each pen color. A "1" in a column indicates that a bit in that color plane is set when the color is drawn, a "0" indicates that a bit is cleared, and a "—" indicates that bits in that color plane are unaffected by drawing (or erasing) with the pen color.

Note

When using the PLOT statements, keep in mind that the optional pen control parameter affects only the up/down movement of the current pen; it does not change the pen color. See the language reference descriptions of the IPLOT, PLOT, and RPLOT statements for further details.

Storing and Loading Display Memory

The HP 98627A's display memory is composed of three "color planes" of 32 768 (=32K) bytes each, for a total of 96 Kbytes of memory. When this memory is displayed, the color planes are visually superimposed; that is, the color of each point on the display is determined by the combination of Red, Green, and Blue dots which are on at that point. Each color dot is turned on when the corresponding bit in that color plane is 1 and turned off when the bit is 0.

In order to store the current display, use the GSTORE statement. This storage of the HP 98627A's display memory into computer memory requires **exactly** 49 152 (=3 × 16 384) INTEGER elements. The following program stores the current display memory in the INTEGER array `Array(*)`.

```
100   PLOTTER IS 28,"98627A" ! Define plotter.
110   !
120   INTEGER Array(1:3,1:16384) ! 49152 elements.
130   GSTORE Array(*)
140   DISP "Display Stored."
150   BEEP
160   WAIT 2
170   !
180   GCLEAR
190   DISP "Display Cleared."
200   BEEP
210   WAIT 2
220   !
230   GLOAD Array(*)
240   DISP "Display Re-Loaded."
250   BEEP
260   !
270   END
```

Refer to GLOAD and GSTORE in the *BASIC Language Reference* manual for more information on loading and storing the display.

The number of pixels in the vertical direction is dependent on the current display format. If the "High Resolution" format is in effect, all array elements are shown. If a smaller display format is in use, only the first elements of each memory plane are displayed. Thus, if the "Standard graphics" format is being used (512 by 390 pixels), only the first 12 480 elements of each primary-color plane are displayed.

The GLOAD statement can be used to load the display. Either a previously stored display or a software-generated numeric array can be used.

Dumping to an External Device

The DUMP GRAPHICS statement or the DUMP GRAPHICS key can be used to dump the screen contents to the current DUMP DEVICE IS device. Each pixel sent to the destination device is represented by the state of one bit, and the state of each bit is determined by performing an inclusive-OR function on corresponding bits of each primary-color memory plane; thus, no color information is sent to the dump device by this type of operation. Examples are as follows.

```
100   PLOTTER IS 28,"HP98627A"
110   DUMP DEVICE IS Printer

400   DUMP GRAPHICS              ! 98627 to Printer.
650   DUMP GRAPHICS #801         ! 98627 to device 801.
810   DUMP GRAPHICS CRT TO #702  !  INTERNAL GRAPHICS to device 702.
```

If EXPANDED is specified in the DUMP DEVICE IS statement, the image is doubled in both X and Y directions before being sent to the destination device. However, if both source and destination are explicitly specified, no expansion is performed. Examples are as follows.

```
100   DUMP DEVICE IS 701,EXPANDED

200   DUMP GRAPHICS              ! Expand and send.
450   DUMP GRAPHICS 28 TO #701   ! Send without expanding.
```

Interactive Graphics

The GRAPHX extension provides the computer with the abilities of accepting inputs from a graphics device and optionally "echoing" these inputs on the graphics output device. The following abilities are provided.

- The ability to digitize individual points on the input (GRAPHICS INPUT IS) device. Each point is selected by first positioning the input device's locator (typically a cursor or stylus) and pressing the "Digitize" button (or the stylus).
- The ability to track the locator's position on the output (PLOTTER IS) device while waiting for a point to be digitized. The locator's position is shown by either a "cross-hair" or by the output device's pen position, depending on the type of plotter device.
- The ability to "continuously" monitor the locator's coordinates.
- The ability to move the output device's cross-hair (or pen), independent of the locator's position.
- The ability to monitor the Digitize button to determine whether or not it is currently being pressed.

Compatible Devices

This software is intended for use with Hewlett-Packard Graphics Language (HPGL) graphics devices. The software currently supports several different types of input and output devices.

Output Devices

At this time, the software can be used with the following HP graphics-output (PLOTTER IS) devices.

- HP 98627A Color Output Interface driving an external color monitor.
- The internal graphics device of Series 200 computers.
- HPGL plotters, in general; examples include the HP 7470 and HP 7580 plotters.

Input Devices

At this time, compatible graphics-input (GRAPHICS INPUT IS) devices include the following:

- HP 9111 Graphics Tablet.
- HP 9874 Digitizer.
- HPGL plotters, in general.
- Keyboards (arrow keys, knob, mouse, etc.)

Setting Up a Session

Using the interactive graphics capabilities requires a few simple program steps and programmer considerations. Both input and output graphics devices must be defined by the program. Your main consideration is how the usable area of the input device maps onto the usable area of the output device, which is discussed in the "Graphics Regions and Mapping" section that follows the first programming example.

Specifying the Output Device

The PLOTTER IS statement is used to specify which device is to be the output device. Several example statements are shown below.

```
PLOTTER IS 28,"98627A"
PLOTTER IS 705,"HPGL"
PLOTTER IS 3,"INTERNAL"
```

Specifying the Input Device

The GRAPHICS INPUT IS statement is used to specify which device is to be used for graphics input. Example statements are shown below.

```
GRAPHICS INPUT IS 706,"HPGL"
GRAPHICS INPUT IS 805,"HPGL"
GRAPHICS INPUT IS KBD,"KBD"
```

Interactive Graphics

The GRAPHX extension provides the computer with the abilities of accepting inputs from a graphics device and optionally "echoing" these inputs on the graphics output device. The following abilities are provided.

- The ability to digitize individual points on the input (GRAPHICS INPUT IS) device. Each point is selected by first positioning the input device's locator (typically a cursor or stylus) and pressing the "Digitize" button (or the stylus).
- The ability to track the locator's position on the output (PLOTTER IS) device while waiting for a point to be digitized. The locator's position is shown by either a "cross-hair" or by the output device's pen position, depending on the type of plotter device.
- The ability to "continuously" monitor the locator's coordinates.
- The ability to move the output device's cross-hair (or pen), independent of the locator's position.
- The ability to monitor the Digitize button to determine whether or not it is currently being pressed.

Compatible Devices

This software is intended for use with Hewlett-Packard Graphics Language (HPGL) graphics devices. The software currently supports several different types of input and output devices.

Output Devices

At this time, the software can be used with the following HP graphics-output (PLOTTER IS) devices.

- HP 98627A Color Output Interface driving an external color monitor.
- The internal graphics device of Series 200 computers.
- HPGL plotters, in general; examples include the HP 7470 and HP 7580 plotters.

Input Devices

At this time, compatible graphics-input (GRAPHICS INPUT IS) devices include the following:

- HP 9111 Graphics Tablet.
- HP 9874 Digitizer.
- HPGL plotters, in general.
- Keyboards (arrow keys, knob, mouse, etc.)

Setting Up a Session

Using the interactive graphics capabilities requires a few simple program steps and programmer considerations. Both input and output graphics devices must be defined by the program. Your main consideration is how the usable area of the input device maps onto the usable area of the output device, which is discussed in the "Graphics Regions and Mapping" section that follows the first programming example.

Specifying the Output Device

The PLOTTER IS statement is used to specify which device is to be the output device. Several example statements are shown below.

```
PLOTTER IS 28,"98627A"
PLOTTER IS 705,"HPGL"
PLOTTER IS 3,"INTERNAL"
```

Specifying the Input Device

The GRAPHICS INPUT IS statement is used to specify which device is to be used for graphics input. Example statements are shown below.

```
GRAPHICS INPUT IS 706,"HPGL"
GRAPHICS INPUT IS 805,"HPGL"
GRAPHICS INPUT IS KBD,"KBD"
```

Interactive Graphics Techniques

The programming techniques in this section show the most common uses of the interactive graphics capabilities. The examples show digitizing single points, digitizing with tracking, and drawing continuous lines.

Digitizing Single Points

"Digitizing a point" simply means that the point's "X" and "Y" coordinates are determined. The point to be digitized is specified by locating the point with the input device's locator and then pressing the Digitize button (or pressing the stylus). The following program shows a simple technique for single-point digitizing. The input device is an HP 9111 Graphics Tablet.

```
100   GINIT                                ! Initialize.
110   GRAPHICS INPUT IS 706,"HPGL"  ! 9111 Graphics Tablet.
120   !
130 Point: DIGITIZE X,Y,Status$  ! Request coords.
140                                ! and status.
150   ! Print coords.
160   DISP USING "#,""X="",K,2X,""Y="",K,2X";X,Y
170   !
180   ! Which region is point in?
190   Region$=Status$[3;1] ! Read 3rd character.
200   SELECT Region$       ! Determine region.
210     CASE ="2"
220       DISP "(Inside VIEWPORT)"
230     CASE ="1"
240       DISP "(Outside VIEWPORT,";
250       DISP " but inside P1,P2)"
260     CASE ="0"
270       DISP "(Outside P1,P2)"
280   END SELECT
290         !
300   GOTO Point
310   !
320   END
```

Executing the DIGITIZE statement causes the following action. The computer first tells the GRAPHICS INPUT IS device that it is to digitize a point. The input device then waits for the Digitize button to be pressed (in this case, for the stylus to be pressed), at which time it determines the point's coordinates. The input device then returns the point's coordinates to the computer. The computer can then execute the next BASIC statement. If no point is selected, the computer will "hang" on the DIGITIZE statement. Pressing CLR I/O will pause the computer, if desired.

The optional string variable (`Status$` in the preceding program) is used to return information about the input device. Here, only the third byte is examined; the value of this byte shows the region in which the point is located. The values 2, 1, and 0 can be returned; the respective definitions are shown in the program.

Graphics Regions and Mapping

As previously mentioned, the **maximum usable area** of a graphics device is bounded by its **hard clip limits**; for example, the pen cannot be made to draw outside these limits on an output device.

The **current usable area** is bounded by the rectangle defined by the points P1 and P2; the **lower-left corner is P1**, and the **upper-right corner is P2**. On many devices, these points can be moved manually or by the program.

When the GINIT statement or a PLOTTER IS statement is executed, points P1 and P2 are read from the plotting device; with GINIT, the plotting device is assumed to be the internal CRT. The value of RATIO is then set to the result of the following calculation:

$$RATIO = (P2x - P1x)/(P2y - P1y)$$

The operating system then automatically scales the P1,P2 rectangle. When GINIT is executed, the following VIEWPORT and WINDOW statements are executed, scaling the rectangle in Graphic Display Units and User Defined Units, respectively.

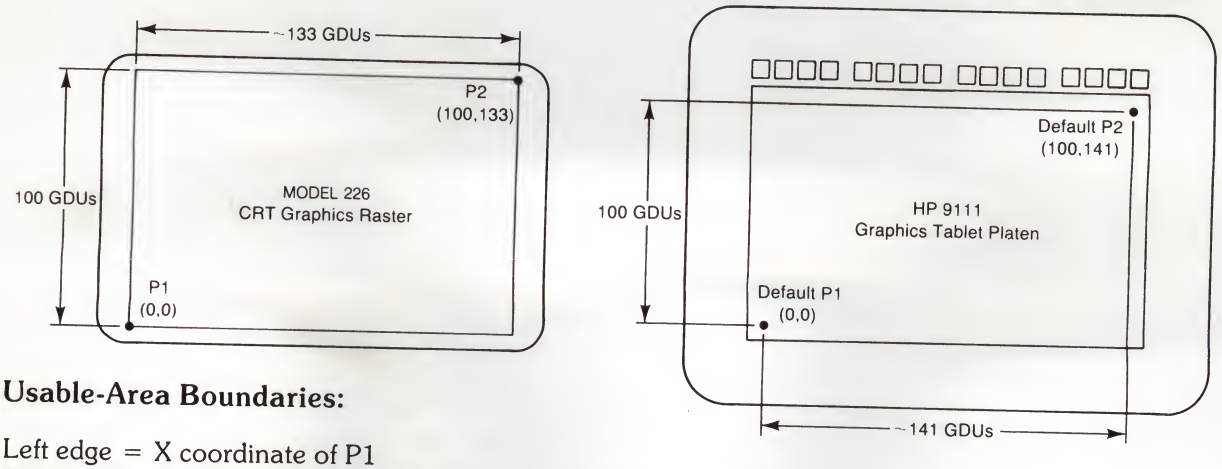
If RATIO >=1:

```
VIEWPORT 0,100*RATIO,0,100
WINDOW 0,100*RATIO,0,100
```

If RATIO < 1:

```
VIEWPORT 0,100,0,100/RATIO
WINDOW 0,100,0,100/RATIO
```

As seen above, the X and Y coordinates of P1 are always both 0 Graphic Display Units. The default Graphic Display Unit coordinates of P2 depend on the device; however, the smaller coordinate of this point is always 100 Graphic Display Units. Two examples are shown below.



Usable-Area Boundaries:

Left edge = X coordinate of P1
Bottom edge = Y coordinate of P1
Right edge = X coordinate of P2
Top edge = Y coordinate of P2

Default Locations of P1 and P2 on the Model 226 and HP 9111

When a PLOTTER IS statement is executed, the locations of points P1 and P2 on the specified device are determined. The current VIEWPORT statement parameters are then used to define the physical area (in GDUs) which is to be scaled (in UDUs) by the WINDOW or SHOW statement currently in effect.

When a subsequent GRAPHICS INPUT IS statement is executed, the operating system attempts to apply the current VIEWPORT and WINDOW (or SHOW) parameters to the P1,P2 rectangle of the input device. In the preceding example, the two usable areas are not identical in size (in Graphic Display Units), since the HP 9111 has a smaller horizontal-to-vertical aspect ratio. The P2 of the input device will automatically be adjusted to equal the GDU's of the output device.

Digitizing with Tracking

The locator may also be "tracked" on the output device while it is being moved (before a point is digitized). Executing the TRACK...IS ON statement accesses this feature, as shown in the following example program.

```
100    GINIT                                ! Restore defaults.
110    GRAPHICS INPUT IS KBD,"KBD"          ! Keyboard is input.
120    PLOTTER IS CRT,"INTERNAL"            ! (Redundant.)
130    TRACK CRT IS On                      ! Enable tracking.
140    !
150    GRAPHICS ON
160    WINDOW -50,50,-20,20          ! Define UDUs.
170    FRAME                         ! Draw bounds.
180    AXES 10,10,0,0,5,5            ! Draw axes.
190    MOVE 0,0                      ! Begin at Origin.
200    !
210    !
220 Track: DIGITIZE X,Y,Status$ ! Request coords.,
230          ! updating cursor until coords. received.
240          !
250          DRAW X,Y                ! Connect points.
260          !
270    GOTO Track
280    !
290    END
```

The TRACK...IS ON statement merely enables the tracking feature; the actual tracking is performed while the DIGITIZE statement is being executed. The locator is "tracked" by moving the output device's "cross-hair" (or pen) correspondingly. Notice that the definition of the DIGITIZE statement has been modified slightly — now its execution causes the locator to be tracked and "echoed" on the output device until the stylus (or Digitize button) is pressed.

After a key satisfying the digitize is pressed, the DIGITIZE statement has finished execution and the DRAW statement is executed. This program draws lines between the digitized points, but you may want to change this response as desired with the appropriate software.

Drawing Continuous Lines

Other common uses of interactive graphics are to draw lines or track the cursor continuously without digitizing. The READ LOCATOR and SET ECHO statements are used in order to determine the input device's locator position and to echo the position on the output device, independent of the Digitize button being pressed. The following program shows an example of implementing this graphics capability in a BASIC program.

```
100    GINIT                                ! Restore defaults.
110    GRAPHICS INPUT IS 706,"HPGL"         ! Define input.
120    PLOTTER IS CRT,"INTERNAL"
130    GRAPHICS ON
140    WINDOW 0,100,0,100                   ! Define UDUs.
150    FRAME                                ! Draw limits.
160    !
170    LOOP
180       READ LOCATOR X,Y,Status$
190       SET ECHO X,Y
200       Button$=Status$[1,1]
210       GOSUB Action
220    END LOOP
230    !
240 Action: IF Button$="0" THEN MOVE X,Y
250         IF Button$="1" THEN DRAW X,Y
260         RETURN
270         !
280    END
```

The preceding program continuously tracks the input locator and monitors the pressed/not-pressed status of the Digitize button (or stylus). The cursor position is continuously echoed on the output device, and lines are drawn if the Digitize button (or stylus point) is pressed.

Interactive Color Graphics

The interactive graphics statements can also be used with the Color Output Interface. The following program shows a typical example of using the interface for this type of graphics application.

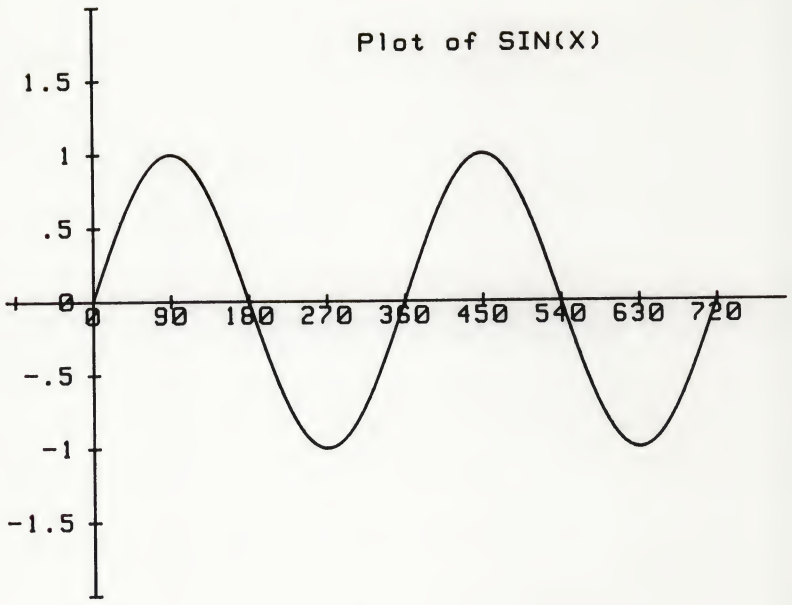
```
100    PLOTTER IS 28,"98627A"          ! Define output.
110    GRAPHICS INPUT IS 706,"HPGL" ! Define input.
120    ! May want to match aspect ratios with VIEWPORT;
130    ! parameters depend on color-display format.
140    ! May also want to move P1,P2 with HPGL commands.
150    !
160    ! Define sfk's.
170    ON KEY 0 LABEL "White " GOSUB White
180    ON KEY 1 LABEL "Red   " GOSUB Red
190    ON KEY 2 LABEL "Yellow" GOSUB Yellow
200    ON KEY 3 LABEL "Green " GOSUB Green
210    ON KEY 5 LABEL "Erase " GOSUB Erase
220    ON KEY 9 LABEL "Clear " GOSUB Clear
230    !
240    LOOP
250       READ LOCATOR X,Y,Status$
260       SET ECHO X,Y
270       Action$=Status$[1,1]
280       GOSUB Action
290       !
300    END LOOP
310    !
320 White:  PEN 1
330         RETURN
340         !
350 Red:    PEN 2
360         RETURN
370         !
380 Yellow: PEN 3
390         RETURN
400         !
410 Green:  PEN 4
420         RETURN
430         !
440 Erase:  PEN -1
450         RETURN
460         !
470 Clear: GCLEAR
480        RETURN
490        !
500 Action: IF Action$="0" THEN MOVE X,Y
510         IF Action$="1" THEN DRAW X,Y
520         RETURN
530         !
540    END
```

The program uses special function keys to change the color of the lines drawn by pressing the stylus onto the platen. The program can be expanded easily to draw and erase all colors, if desired.

Example Graphics Programs

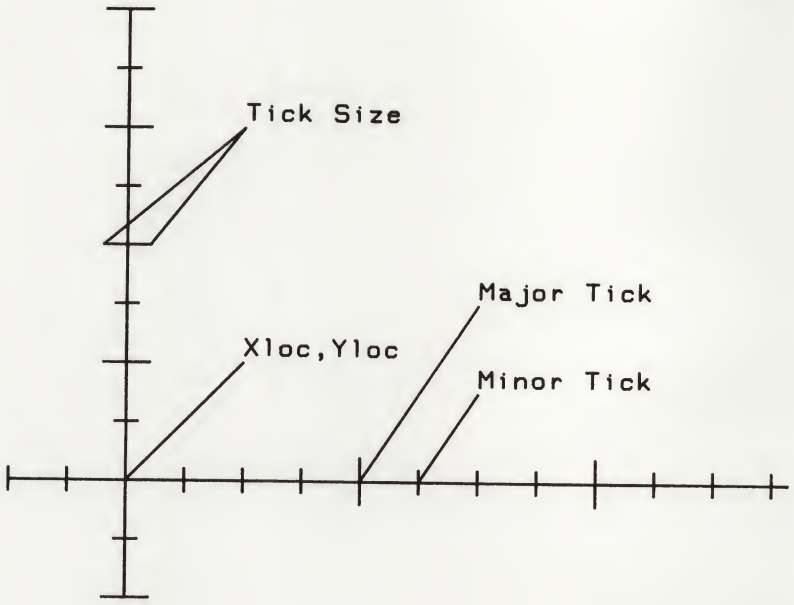
The following programs use graphics to help illustrate the operation of several of the graphics statements available in BASIC. You may wish to modify or entirely rewrite the programs to better understand how the statements work.

Sine



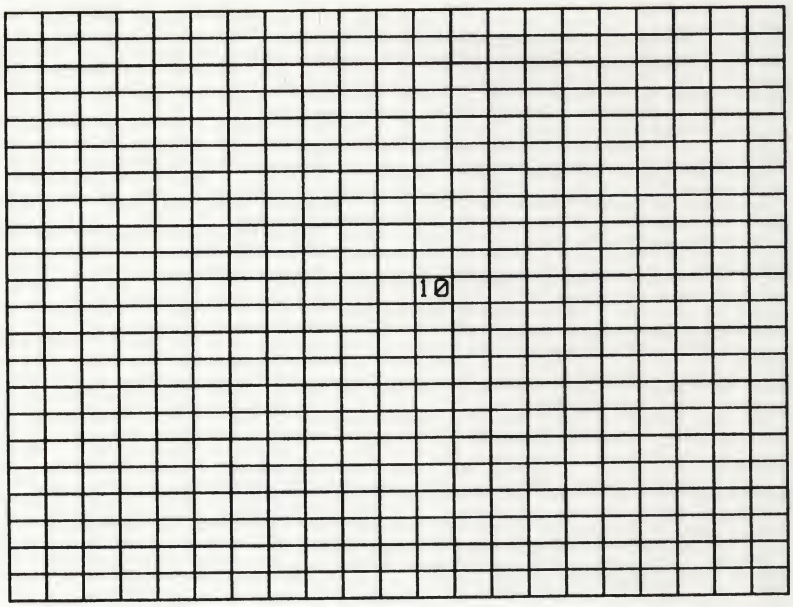

```
10    ! Program: SINE
20    !
30    ! Shows some basics of drawing and labeling.
40    !
50    DEG                              ! DEGREES
60    GINIT                            ! INITIALIZE
70    GRAPHICS ON                      ! RASTER ON
80    PRINT CHR$(12);                  ! CLEAR ALPHA
90    WINDOW -100,800,-2,2             ! SET WINDOW
100   AXES 90,.5                       ! DRAW AXES
110   !
120   LORG 6                           ! LABEL X AXIS
130   FOR I=0 TO 720 STEP 90
140   MOVE I,0
150   LABEL I
160   NEXT I
170   !
180   LORG 8                           ! LABEL Y AXIS
190   FOR I=-1.5 TO 1.5 STEP .5
200   MOVE 0,I
210   LABEL I
220   NEXT I
230   !
240   LORG 5                           ! LABEL PLOT
250   MOVE 450,1.75
260   LABEL "Plot of SIN(X)"
270   !
280   MOVE 0,0                         ! PLOT SINE
290   FOR X=0 TO 720
300   DRAW X,SIN(X)
310   NEXT X
320   !
330   END
```

Axes




```
10    ! Program: AXES
20    !
30    ! Draw and label the AXES statement.
40    !
50    GINIT
60    GRAPHICS ON
70    ALPHA OFF
80    !
90    Xloc=20                    ! X AXIS LOCATION
100   Yloc=20                    ! Y AXIS LOCATION
110   Xmaj=4                     ! MAJOR TICK COUNT
120   Ymaj=2                     ! MAJOR TICK COUNT
130   Size=8                     ! LENGTH OF TICKS
140   !
150   FOR I=100 TO 10 STEP -1
160   PEN -1
170   AXES Xtic,Ytic,Xloc,Yloc,Xmaj,Ymaj,Size
180   Xtic=I
190   Ytic=I
200   PEN 1
210   AXES Xtic,Ytic,Xloc,Yloc,Xmaj,Ymaj,Size
220   NEXT I
230   !
240   MOVE Xloc,Yloc             ! LABEL THE AXES
250   IDRAW 20,20
260   LABEL "Xloc,Yloc"
270   MOVE Xloc+40,Yloc
280   IDRAW 20,30
290   LABEL "Major Tick"
300   MOVE Xloc+50,Yloc
310   IDRAW 10,15
320   LABEL "Minor Tick"
330   MOVE Xloc-Size/2,Yloc+40
340   DRAW 40,80
350   MOVE Xloc+Size/2,Yloc+40
360   DRAW 40,80
370   LABEL "Tick Size"
380   !
390   END
```

Grid



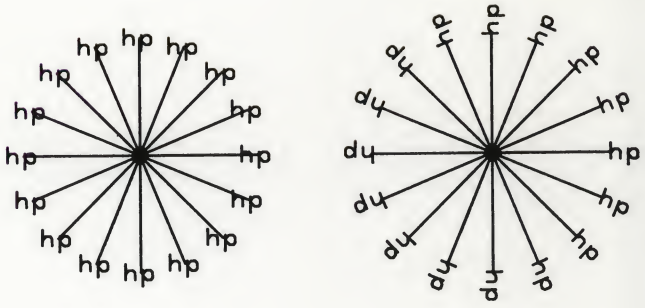

```
10    ! Program: GRID
20    !
30    ! Shows various size grids.
40    !
50    GINIT
60    GRAPHICS ON
70    PRINT CHR$(12);
80    !
90    WINDOW -110,100,-110,110
100   !
110   Yloc=0                  ! CENTER AT 0,0
120   Xloc=0
130   Xmaj=6
140   Ymaj=2
150   Size=20
160   !
170   LORG 4
180   !
190   FOR I=10 TO 100 STEP 2
200     Xtic=I
210     Ytic=I
220     GCLEAR
230     MOVE I/2,0
240     LABEL I
250      GRID Xtic,Ytic,Xloc,Yloc,Xmaj,Ymaj,Size
260     WAIT (100-I)/100
270   NEXT I
280   !
290   WAIT 2
300   GRAPHICS OFF
310   END
```

Csize




```
10    ! Program: CSIZE
20    !
30    !  VARIABLE SIZE CHARACTERS
40    !
50    ON KEY 0 LABEL " HEIGHT" GOSUB Flag_off
60    ON KEY 1 LABEL " RATIO" GOSUB Flag_on
70    ON KEY 4 LABEL " QUIT" GOTO Quit
80    PRINT CHR$(12);
90    DISP "ROTATE KNOB TO CHANGE SIZE.";
100   ON KNOB .2 GOTO Change
110   !
120   H=5                        ! HEIGHT
130   W=.6                       ! HEIGHT/WIDTH RATIO
140   GOTO Size
150   !
160 Change: ! CHANGE THE CHARACTER SIZE
170   IF Flag THEN
180     W=W+KNOBX/100
190     IF W<.01 AND W>0 THEN W=0
200     ELSE
210     H=H+KNOBX/10
220   END IF
230   DISP
240   DISP USING "2X,3A,3D.D";"H =";H;"R =";W
250   !
260 Size: GINIT
270       GRAPHICS ON
280       CSIZE H,W
290       MOVE 67,50
300       LORG 5
310       LABEL "SIN(X)"
320       GOTO 320             ! WAIT HERE
330   !
340 Flag_on:   Flag=1
350            RETURN
360   !
370 Flag_off:  Flag=0
380            RETURN
390   !
400 Quit: GRAPHICS OFF
410       OUTPUT 2 USING "#,B";255,75
420       END
```

Label



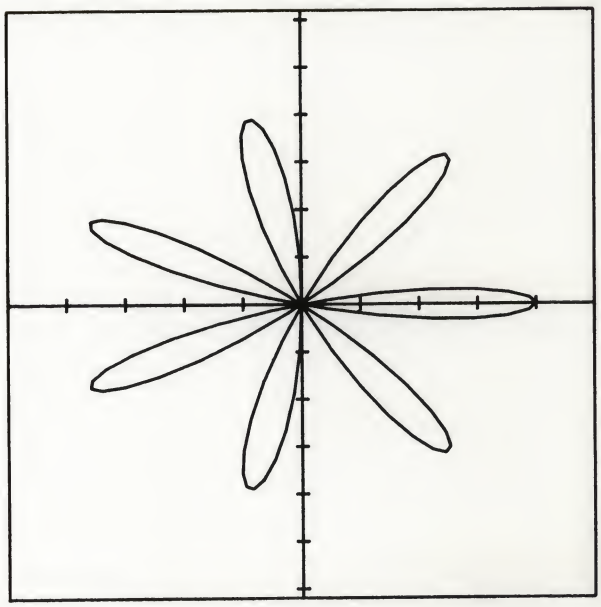
```
10    ! Program: LABEL
20    !
30    DEG
40    GINIT
50    GRAPHICS ON
60    Clear_crt$=CHR$(255)&CHR$(75)
70    OUTPUT 2;Clear_crt$;
80    SHOW -100,100,-100,100
90    !
100   FOR I=0 TO 360 STEP 22.5 ! NON-ROTATED
110     MOVE -60,0
120     PIVOT I
130     IDRAW 40,0
140     LORG 5
150     LABEL "hp"
160   NEXT I
170   !
180   FOR I=0 TO 360 STEP 22.5  ! ROTATED
190     MOVE 60,0
200     PIVOT I
210     IDRAW 40,0
220     LORG 2
230     LDIR I                  ! NOTE LDIR USED
240     LABEL "hp"
250   NEXT I
260   !
270   END
```

Ginit

Reverse Graphics

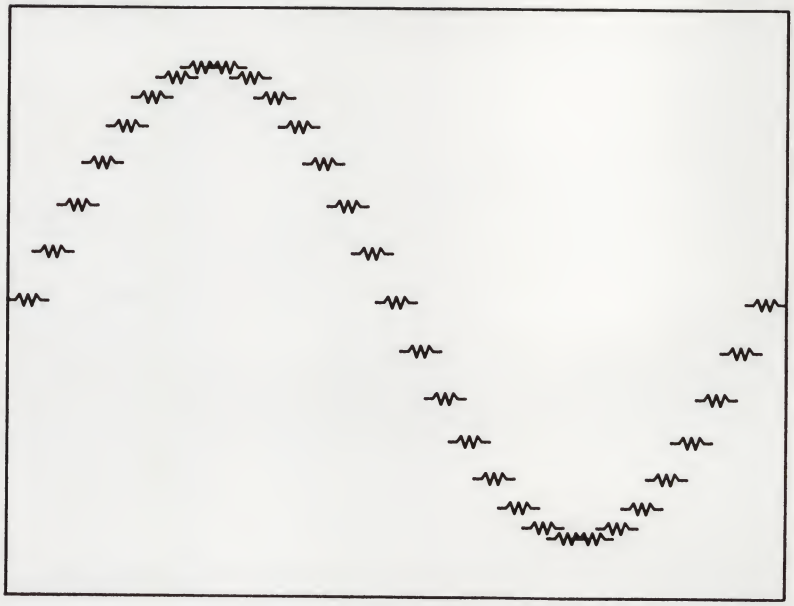
```
10    ! Program: GINIT
20    !
30    ! This program does a specialized 'GINIT'
40    !
50    PRINT CHR$(12); ! CLEAR SCREEN
60    !
70    ! CUSTOMIZED 'GINIT'
80    !
90    PLOTTER IS 3,"INTERNAL"
100   CLIP OFF
110   PIVOT 0
120   PEN 1
130   LINE TYPE 1,5
140   LORG 5
150   CSIZE 8,-.6
160   LDIR 0
170   MOVE 90,60
180   GCLEAR
190   VIEWPORT 0,RATIO*100,0,100
200   WINDOW 0,RATIO*100,0,100
210   !
220   GRAPHICS ON
230   LABEL "Reverse Graphics"
240   LIST 60,230
250   END
```

Rose



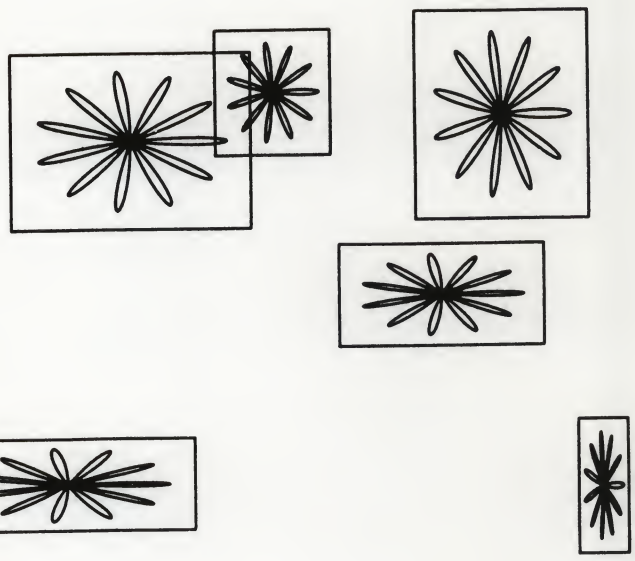
```
10     ! Program: ROSE
20     !
30     ! Rotate the knob, change the view.
40     !
50     DEG
60     GINIT
70     GRAPHICS ON
80     PRINT CHR$(12);
90     ON KNOB ,1 GOTO Scale
100    !
110    Max=40    ! starting value
120    !
130 Scale:  Max=Max+KNOBX
140    IF Max<1 THEN Max=1
150    DISP Max
160    !
170    IF Max>72 THEN
180      Dx=Max-72
190      SHOW -51+Dx,50-Dx,-63+Dx,62-Dx
200     ELSE
210      VIEWPORT 62-Max,62+Max,50-Max,50+Max
220      WINDOW -50,50,-62,62
230    END IF
240    GCLEAR
250    FRAME
260    AXES 10,10,0,0
270    FOR I=0 TO 180 STEP 2
280      P=40*COS(7*I)
290      X=P*COS(I)
300      Y=P*SIN(I)
310      IF I=0 THEN MOVE X,Y
320      DRAW X,Y
330    NEXT I
340    !
350    GOTO 170
360    END
```

Rplot



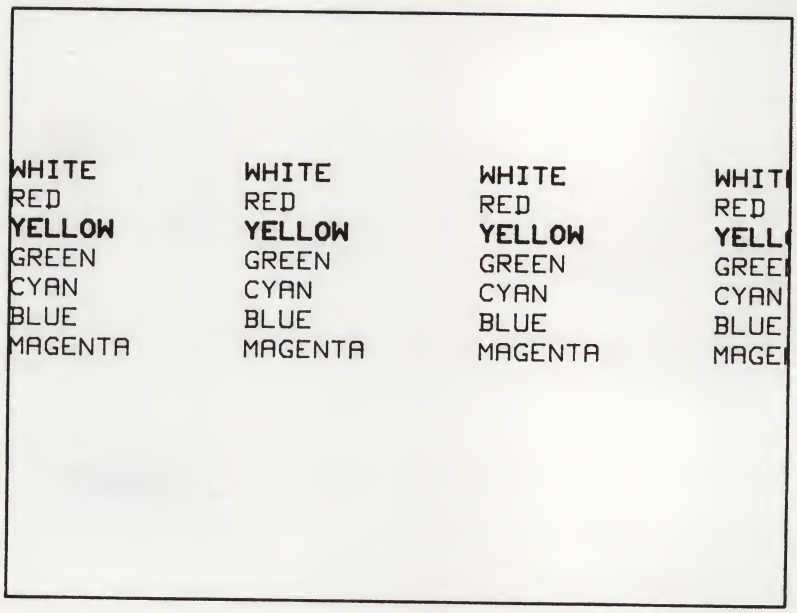
```
10     ! Program: RPLOT
20     !
30     ! Repeats an image at various locations.
40     !
50     DEG
60     GINIT
70     GRAPHICS ON
80     WINDOW -10,370,-100,100
90     PRINT CHR$(12);    ! CLEAR SCREEN
100    DISP " RPLOT"
110    FRAME
120    !
130    FOR I=0 TO 360 STEP 12
140      MOVE I,SIN(I)*80
150      GOSUB Shape
160    NEXT I
170    !
180    GOTO Quit
190    !
200 Shape:    ! DRAW A RESISTER
210    !
220    RPLOT -10,0,1
230    RPLOT -6,0
240    RPLOT -4,2
250    RPLOT -2,-2
260    RPLOT 0,2
270    RPLOT 2,-2
280    RPLOT 4,2
290    RPLOT 6,0
300    RPLOT 10,0
310    RETURN
320    !
330 Quit:    END
```

Randomview



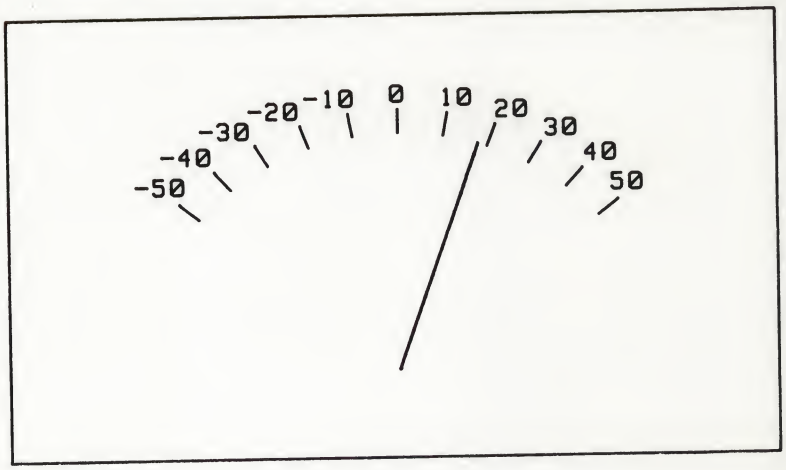
```
10     ! Program: RANDOMVIEW
20     !
30     RANDOMIZE
40     !
50 Start: ! Demonstration of VIEWPORT and WINDOW
60     !
70     DEG
80     GINIT
90     GRAPHICS ON
100    ALPHA OFF
110    !
120    ! Generate some random numbers
130    !
140    Xmin=RND*131
150    Xmax=Xmin+RND*(131-Xmin)
160    Ymin=RND*100
170    Ymax=Ymin+RND*(100-Ymin)
180    !
190    ! Set VIEWPORT to random area
200    !
210    VIEWPORT Xmin,Xmax,Ymin,Ymax
220    WINDOW -50,50,-50,50
230    FRAME
240    !
250    ! Draw a rose within the area
260    !
270    FOR I=0 TO 200
280      P=40*COS(11*I)          ! ELEVEN LEAF ROSE
290      X=P*COS(I)
300      Y=P*SIN(I)
310      DISP INT(Xmax-Xmin);":";INT(Ymax-Ymin)
320       IF I=0 THEN MOVE X,Y
330      DRAW X,Y
340    NEXT I
350    !
360    GOTO Start                ! DO IT AGAIN
370    END
```

Color



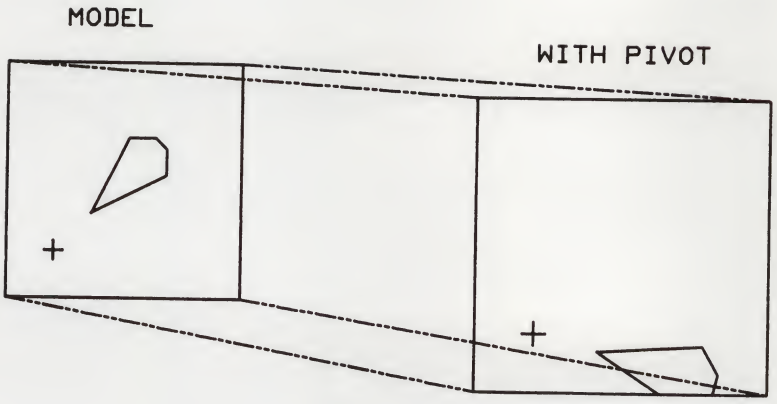

```
10    ! Program: COLOR
20    !
30    ! This program works with the 98627A
40    ! Color Output Interface
50    !
60    ! Note that a 'PLOTTER IS' statement must
70    ! immediately follow 'GINIT' statement.
80    !
90    ! Note different pen assignments.
100   !
110   GINIT
120   PLOTTER IS 28,"98627A"
130   GRAPHICS ON
140   PEN 1
150   FRAME
160   !
170   FOR X=0 TO 120 STEP 40
180   MOVE X,70
190   PEN 1
200   LABEL "WHITE"
210   PEN 2
220   LABEL "RED"
230   PEN 3
240   LABEL "YELLOW"
250   PEN 4
260   LABEL "GREEN"
270   PEN 5
280   LABEL "CYAN"
290   PEN 6
300   LABEL "BLUE"
310   PEN 7
320   LABEL "MAGENTA"
330   NEXT X
340   END
```

Meter




```
10    ! Program: METER
20    !
30    ! Simulation of a analog meter
40    ! KNOB and RANDOM needle movement.
50    !
60    DEG
70    GINIT
80    GRAPHICS ON
90    CLIP 0,131,25,100
100   WINDOW -150,150,-100,150
110   OUTPUT 2 USING "#,B";255,75  ! CLEAR SCREEN
120   !
130   FRAME                        ! CREATE METER
140   FOR Tic=-50 TO 50 STEP 10
150     MOVE 0,0
160     PIVOT Tic
170     LORG 4
180     MOVE 0,100
190     IDRAW 0,10
200     LABEL -Tic
210   NEXT Tic
220   !
230   ON KNOB .2 GOTO Meter
240   !
250 Meter: ! GENERATE RANDOM METER MOVEMENT
260   PEN -1
270   MOVE 0,0
280   DRAW 100,0                   ! UNDRAW NEEDLE
290   P=(P+4*RND-2) MOD 60+KNOBX/10
300   MOVE 0,0
310   PIVOT 90-P
320   DISP INT(P)
330   PEN 1
340   MOVE 0,0
350   DRAW 100,0                   ! REDRAW NEEDLE
360   WAIT .07
370   GOTO Meter
380   !
390   END
```

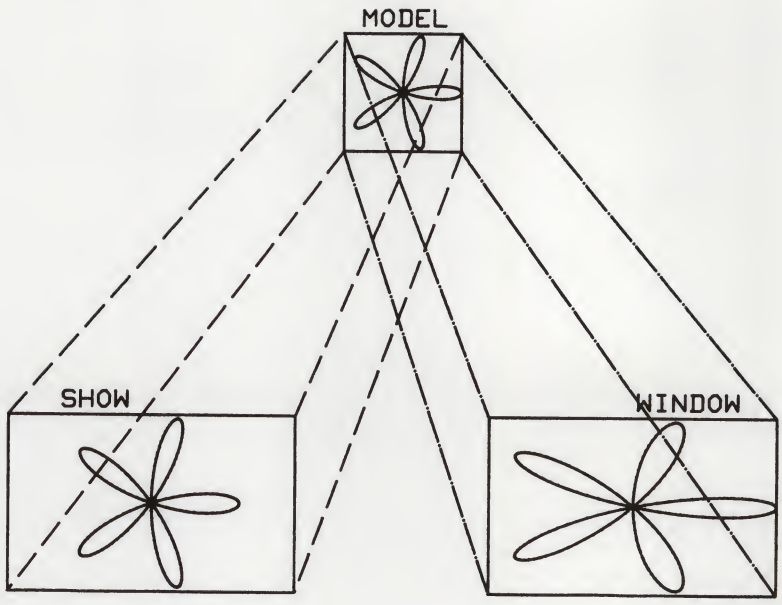
Pivot




```
10    ! Program: PIVOT
20    !
30    ! Shows pivoting around a point.
40    !
50    DEG
60    GINIT
70    GRAPHICS ON
80    PRINT CHR$(12);
90    DIM X(4),Y(4)
100   DATA 40,20,0,14,-6,6,-14,0,-20,-40 ! SHAPE
110   FOR I=0 TO 4
120     READ X(I),Y(I)
130   NEXT I
140   !
150   DATA 80,130,35,85,0,40,50,90 ! 'WINDOWS'
160   READ Sl,Sr,Sb,St,Ml,Mr,Mb,Mt
170   !
180   DIM Orgx(3),Orgy(3)
190   DATA 40,60,40,40,20,20,0,0 ! ORIGINS
200   FOR I=0 TO 3
210     READ Orgx(I),Orgy(I)
220   NEXT I
230   MOVE 10,95
240   LABEL "MODEL"
250   MOVE 90,90
260   LABEL "WITH PIVOT"
270   LINE TYPE 8
280   MOVE Ml,Mb                    ! CONNECT LINES
290   DRAW Sl,Sb
300   MOVE Mr,Mb
310   DRAW Sr,Sb
320   MOVE Mr,Mt
330   DRAW Sr,St
340   MOVE Sl,St
350   DRAW Ml,Mt
360   !
370   MOVE Ml,Mt
380   P=1
390   LINE TYPE 1
400   Ox=Orgx(Index)
410   Oy=Orgy(Index)
420   GOSUB Model
430   !
```

```
440    VIEWPORT Sl,Sr,Sb,St
450    SHOW -25,100,-25,100
460    GOSUB Shape
470    DISP "Angle =";Angle
480    Angle=Angle+5
490    IF Angle<361 THEN 460
500    CALL Cursor(Ox,Oy,-1)          ! PIVOT POINT
510    P=-1
520    GOSUB Model
530    Angle=0
540    Index=Index+1
550    IF Index>3 THEN Quit
560    GOTO 380
570    !
580 Model:  VIEWPORT Ml,Mr,Mb,Mt
590         SHOW -25,100,-25,100
600         FRAME
610         GOSUB Shape
620         RETURN
630 Shape:  ! DRAW IN CURRENT 'WINDOW'
640         PEN -1
650         MOVE 20,20
660         FOR I=0 TO 4
670         IDRAW X(I),Y(I)
680         NEXT I
690         MOVE Ox,Oy
700         PEN 1
710         CALL Cursor(Ox,Oy,P)
720         PIVOT Angle
730         PEN 1
740         FRAME
750         MOVE 20,20
760         FOR I=0 TO 4
770         IDRAW X(I),Y(I)
780         NEXT I
790         RETURN
800 Quit:DISP
810      END
820      !
830      ! ------- SUB PROGRAM ---------
840      !
850 SUB Cursor(X,Y,P)
860     PEN P
870     PIVOT 0
880     MOVE X,Y
890     IMOVE 5,0
900     IDRAW -10,0
910     IMOVE 5,5
920     IDRAW 0,-10
930     MOVE X,Y
940     SUBEXIT
950 SUBEND
```

Showwindow




```
10    ! PROGRAM: SHOWWINDOW
20    !
30    ! Compares the mapping of SHOW and WINDOW
40    !
50    DEG
60    GINIT
70    GRAPHICS ON
80    DISP "WINDOW AND SHOW"
90    DIM X(200),Y(200),Center$[40]
100   FOR I=0 TO 180
110   R=100*COS(5*I)
120   X(I)=R*COS(I)
130   Y(I)=R*SIN(I)
140   NEXT I
150   !
160   ! Determine if running on a 9826 or 9836.
170   ! Needed for centering of prompt.
180   IF RATIO>1.32 THEN
190     Space=20
200    ELSE
210     Space=35
220   END IF
230   FOR I=1 TO Space
240     Center$=Center$&" "
250   NEXT I
260   !
270   !
280   DATA 0,30,0,50,57,77,75,95
290   READ Sl,Sr,Sb,St,Ml,Mr,Mb,Mt
300   !
310 Loop: ! FRAME THE PLOTTING AREAS
320   GINIT
330   PRINT CHR$(12);
340   ALPHA OFF
350   Wr=131
360   Wl=Wr-Sr
370   Wt=St
380   Wb=Sb
390   VIEWPORT Sl,Sr,Sb,St
400   FRAME
410   VIEWPORT Ml,Mr,Mb,Mt
420   FRAME
430   VIEWPORT Wl,Wr,Wb,Wt
```

```
440   FRAME
450   VIEWPORT 0,131.36,0,100
460   !
470   LINE TYPE 5                ! 'SHOW' LINES
480   MOVE Ml,Mb
490   DRAW Sl,Sb
500   MOVE Mr,Mb
510   DRAW Sr,Sb
520   MOVE Mr,Mt
530   DRAW Sr,St
540   MOVE Sl,St
550   DRAW Ml,Mt
560   LINE TYPE 6                ! 'WINDOW' LINES
570   MOVE Ml,Mb
580   DRAW Wl,Wb
590   MOVE Mr,Mb
600   DRAW Wr,Wb
610   MOVE Mr,Mt
620   DRAW Wr,Wt
630   MOVE Ml,Mt
640   DRAW Wl,Wt
650   !
660   LINE TYPE 1
670   VIEWPORT 0,131.36,0,100 ! LABELS
680   WINDOW 0,131.36,0,100
690   LORG 1
700   MOVE Ml,Mt
710   LABEL "MODEL"
720   MOVE Sl,St
730   LORG 1
740   LABEL "SHOW"
750   MOVE Wr,Wt
760   LORG 7
770   LABEL "WINDOW"
780   !
790   FOR I=1 TO 180             ! DRAW A ROSE IN THREE
800   LINE TYPE 1                ! PLACES AT ONCE
810   VIEWPORT Ml,Mr,Mb,Mt
820   SHOW -100,100,-100,100
830   MOVE X(I-1),Y(I-1)
840   DRAW X(I),Y(I)
850   VIEWPORT Sl,Sr,Sb,St
860   SHOW -100,100,-100,100
870   MOVE X(I-1),Y(I-1)
880   DRAW X(I),Y(I)
890   VIEWPORT Wl,Wr,Wb,Wt
900   WINDOW -100,100,-100,100
910   MOVE X(I-1),Y(I-1)
920   DRAW X(I),Y(I)
930   NEXT I
940   !
950   DISP Center$;"NEW RATIO";
960   OUTPUT 2;Center$;
970   INPUT "",Ra
980   IF Ra<=0 THEN Quit
990   IF Ra>=1 THEN
1000    Sr=49
1010    St=60/Ra
1020   ELSE
1030    St=60
1040    Sr=49*Ra
1050  END IF
1060  GOTO Loop
1070  !
1080 Quit:GRAPHICS OFF
1090      OUTPUT 2 USING "#,B";255,75
1100      END
```

Gload



1 2 3 4

```
10    ! Program: GLOAD
20    !
30    ! DIMENSION  'PICTURE' ARRAYS
40    !
50    OPTION BASE 1
60    A=7500                          ! ARRAY SIZE FOR 9826
70    IF RATIO<1.32 THEN A=12480  ! ARRAY SIZE FOR 9836
80    ALLOCATE INTEGER P1(A),P2(A),P3(A),P4(A)
90    !
100   DEG
110   GINIT
120   GRAPHICS ON
130   PRINT CHR$(12);
140   DISP "GLOAD/GSTORE"
150   SHOW -800,800,-800,800
160   !
170   GCLEAR
180   FOR I=0 TO 90 STEP .5      ! HEAD
190     MOVE 0,0
200     PIVOT I
210     MOVE -200,0
220     DRAW 200,0
230     MOVE 0,0
240     PIVOT -I
250     MOVE -200,0
260     DRAW 200,0
270   NEXT I
280   !
290   PEN -1
300   FOR I=0 TO 360 STEP 8        ! EYE
310     MOVE 40,100
320     PIVOT I
330     IDRAW 10,0
340     NEXT I
350   !
360   PEN 1
370   MOVE 300,0
380   LABEL "1"
390   GSTORE P1(*) ! SAVE PICTURE IN ARRAY 'P1'
400   BEEP 3400,.02
410   PEN -1
420   MOVE 300,0
430   LABEL "1"
440   !
```

```
450   A=0
460   B=10
470   GOSUB Mouth
480   MOVE 400,0
490   LABEL "2"
500   GSTORE P2(*) ! SAVE PICTURE IN ARRAY 'P2'
510   BEEP 3400,.02
520   PEN -1
530   MOVE 400,0
540   LABEL "2"
550   !
560   A=10
570   B=20
580   GOSUB Mouth
590   MOVE 500,0
600   LABEL "3"
610   GSTORE P3(*) ! SAVE PICTURE IN ARRAY 'P3'
620   BEEP 3400,.02
630   PEN -1
640   MOVE 500,0
650   LABEL "3"
660   !
670   A=20
680   B=40
690   GOSUB Mouth
700   MOVE 600,0
710   LABEL "4"
720   GSTORE P4(*) ! SAVE PICTURE IN ARRAY 'P4'
730   BEEP 3400,.02
740   PEN -1
750   MOVE 600,0
760   LABEL "4"
770   !
780 Movie: ! animation loop
790        GLOAD P1(*)
800        GLOAD P2(*)
810        GLOAD P3(*)
820        GLOAD P4(*)
830        GOTO Movie
840        !
850 Mouth:  PEN -1
860         FOR I=A TO B STEP .2
870         MOVE 0,0
880         PIVOT I
890         DRAW 200,0
900         MOVE 0,0
910         PIVOT -I
920         DRAW 200,0
930         NEXT I
940         PEN 1
950         PIVOT 0
960         RETURN
970    END
```

Notes

Transporting Programs

Chapter 15

If you have programs which were written on previous Series 200 BASIC systems, you can use these same programs with little or no changes. This chapter describes the differences between BASIC 2.0/2.1 extensions and BASIC 3.0.

In general, you can LOAD and RUN BASIC 2.0 programs on BASIC 3.0. The major task you have to perform is configure the BASIC 3.0 system with the necessary BIN files. If a program uses CSUBs or the PHYREC binary, some additional work is required.

Configuring BASIC

BASIC 3.0 is contained in one SYSTM file, several language extension BIN files and several driver BIN files. When you configure your system, you choose which files you want to load. There are three ways you can do this.

1. After booting the BASIC 3.0 SYSTM file, load a program and try to run it. Error 1 indicates which language extension BIN files are missing. If you LOAD BIN the ERR extension, the name of the missing file is given. If the ERR extension is not loaded, the number of the missing file is given. These numbers are listed in the Useful Tables Section of the *BASIC Language Reference*. When the error occurs, LOAD BIN the needed BIN file and re-run the program.

   If the program uses any interfaces, you must also load the appropriate drivers. An I/O error indicates when an interface driver is needed. A mass storage error indicates when a mass storage device is needed.

2. Use the Language History table at the beginning of the *BASIC Language Reference Manual*. Look up the statements your program uses and see which BINs you need to load. For example, if your program uses the MAT statement which was located in AP2.0, you now need to load the MAT BIN.

   Review the table of device types in the MASS STORAGE IS section of the *BASIC Language Reference Manual*, or the interface table in the Configuration chapter of the *BASIC User's Guide*. Determine which drivers you need and load them.

3. The program CONFIGURE on the language extension and drivers discs can help you select the BIN files you need and configures the system for you.

Once you have loaded all the BIN files you need and your program runs, you can store the system. Use the STORE SYSTEM statement. This statement is described in both the *BASIC User's Guide* and in this manual in Chapter 2. A STORE'd system requires a higher capacity mass storage device than the Model 226 or 236 internal drive.

Statement Changes

There are several statements added with BASIC 3.0. These are listed below.

KNOBY	PRINTER IS file
LIST BIN	READ LABEL
MAXREAL	RES
MINREAL	SCRATCH BIN
MODULO	SECURE
PDIR	SET LOCATOR
PLOTTER IS file	STORE SYSTEM
PRINT LABEL	SYSBOOT

One statement was deleted, RE-STORE BIN.

KNOBY

While most of these statements are enhancements which you can chose to use or not, the statement KNOBY does effect how a program interprets the rotation of the knob.

KNOBX returns the number of horizontal knob pulses. KNOBY returns the number of vertical knob pulses. In previous BASIC versions, KNOBX returned the total number of pulses without regard for vertical or horizontal movement.

An example program of using KNOBX and KNOBY is included in the "Program Structure and Flow" chapter of this manual, in the section Using the Knob. If you have the BASIC 2.0 manual, you can compare the programs.

CSUBs

If you used Pascal compiled subprograms (CSUBs) in your BASIC 2.0 programs, you will have to purchase a Pascal 3.0 system upgrade and a CSUB Utility upgrade to use those CSUBs with BASIC 3.0. You recompile the Pascal routine on Pascal 3.0 and re-execute the CSUB utility to make the routine look like a BASIC subprogram.

PHYREC

The PHYREC routine which allowed you to read from and write to physical records on a disc is changed from a binary program to a CSUB with BASIC 3.0. The PHYREC CSUB is located on the BASIC Utilities Disc 1.

You must append the PHYREC CSUB to your program and change PHYREAD/PHYWRITE statements. If the PHYREC binary is appended to a program, a warning message is displayed and the binary is ignore by BASIC 3.0.

Use the followig steps to locate all the lines for an application which uses PHYREC, change them to call the PHYREC CSUB and to append the PHYREC CSUB.

1. Boot a BASIC 2.0 system.
2. Delete the PHYREC binary.

```
LOAD "program"
SAVE "program2" - This saves the program without the binary.
SCRATCH A       - This deletes the program and binary from memory.
GET "program2"  - Calls to PHYREC are commented. Write down the line num-
                  bers.
RE-STORE "program"
PURGE "program2"
```

3. Attach the PHYREC CSUB.

```
LOADSUB ALL FROM "PHYREC"
```

   This file is located on Disc 1 of the BASIC 3.0 Utilities Library. **Do not try to run your application until you have completed all steps.**
4. Uncomment and change all the calls to PHYREC. These are the lines you noted in step 2 above.

```
PHYREAD Sector,Int_array(*)  > Phyread(Sector,Int_array(*))
PHYWRITE Sector,Int_array(*) > Phywrite(Sector,Int_array(*))
```

5. If `Sector` is declared to be an INTEGER, you need to put it into parentheses so that PHYREC will interpret it as a REAL.

```
Phyread((Sector),Int_array(*))
```

6. The syntax for a conditional call must be changed from:

```
IF condition THEN PHYREAD
```

   to:

```
IF condition THEN
   Phyread(Sector,Int_array(*))
END IF
```

   or to:

```
IF condition THEN CALL Phyread(Sector,Int_array(*))
```

7. `RE-STORE "program"` after you have completed the changes.
8. Boot BASIC 3.0 and run your application.

Hardware

BASIC 3.0 supports hardware which was not available with BASIC 2.0. The new hardware includes the HP 46020A keyboard and the HP 9837 with bit-mapped display.

If you are changing hardware, or your application runs on more than one Series 200 computer, you need to check any statement which interfaces with the keyboard or display. Check for information on the effects of different hardware on a particular statement in the *BASIC Language Reference Manual* under that statement.

Error Messages

1 Missing option or configuration error. If a statement requires an option which is not loaded, the option number or option name is given along with error 1. These numbers are listed in the Useful Tables section. Error 1 without an option number indicates other configuration errors.

2 Memory overflow. If you get this error while loading a file, the program is too large for the computer's memory. If the program loads, but you get this error when you press RUN, then the overflow was caused by the variable declarations. Either way, you need to modify the program or add more read/write memory.

3 Line not found in current context. Could be a GOTO or GOSUB that references a non-existent (or deleted) line, or an EDIT command that refers to a non-existent line label.

4 Improper RETURN. Executing a RETURN statement without previously executing an appropriate GOSUB or function call. Also, a RETURN statement in a user-defined function with no value specified.

5 Improper context terminator. You forgot to put an END statement in the program. Also applies to SUBEND and FNEND.

6 Improper FOR...NEXT matching. Executing a NEXT statement without previously executing the matching FOR statement. Indicates improper nesting or overlapping of the loops.

7 Undefined function or subprogram. Attempt to call a SUB or user-defined function that is not in memory. Look out for program lines that assumed an optional CALL.

8 Improper parameter matching. A type mismatch between a pass parameter and a formal parameter of a subprogram.

9 Improper number of parameters. Passing either too few or too many parameters to a subprogram. Applies only to non-optional parameters.

10 String type required. Attempting to return a numeric from a user-defined string function.

11 Numeric type required. Attempting to return a string from a user-defined numeric function.

12 Attempt to redeclare variable. Including the same variable name twice in declarative statements such as DIM or INTEGER.

13 Array dimensions not specified. Using the `(*)` symbol after a variable name when that variable has never been declared as an array.

14 OPTION BASE not allowed here. The OPTION BASE statement must appear before any declarative statements such as DIM or INTEGER. Only one OPTION BASE statement is allowed in one context.

15 Invalid bounds. Attempt to declare an array with more than 32 767 elements or with upper bound less than lower bound.

16 Improper or inconsistent dimensions. Using the wrong number of subscripts when referencing an array element.

17 Subscript out of range. A subscript in an array reference is outside the current bounds of the array.

18 String overflow or substring error. String overflow is an attempt to put too many characters into a string (exceeding dimensioned length). This can happen in an assignment, an ENTER an INPUT, or a READ. A substring error is an attempted violation of the rules for substrings. Watch out for null strings where you weren't expecting them.

19 Improper value or out of range. A value is too large or too small. Applies to items found in a variety of statements. Often occurs when the number builder overflows (or underflows) during an I/O operation.

20 INTEGER overflow. An assignment or result exceeds the range allowed for INTEGER variables. Must be −32 768 thru 32 767.

22 REAL overflow. An assignment or result exceeds the range allowed for REAL variables.

24 Trig argument too large for accurate evaluation. Out-of-range argument for a function such as TAN or LDIR.

25 Magnitude of ASN or ACS argument is greater than 1. Arguments to these functions must be in the range −1 thru +1.

26 Zero to non-positive power. Exponentiation error.

27 Negative base to non-integer power. Exponentiation error.

28 LOG or LGT of a non-positive number.

29 Illegal floating point number. Does not occur as a result of any calculations, but is possible when a FORMAT OFF I/O operation fills a REAL variable with something other than a REAL number.

30 SQR of a negative number.

31 Division (or MOD) by zero.

32 String does not represent a valid number. Attempt to use "non-numeric" characters as an argument for VAL, data for a READ, or in response to an INPUT statement requesting a number.

33 Improper argument for NUM or RPT$. Null string not allowed.

34 Referenced line not an IMAGE statement. A USING clause contains a line identifier, and the line referred to is not an IMAGE statement.

35 Improper image. See IMAGE or the appropriate keyword in the *BASIC Language Reference.*

36 Out of data in READ. A READ statement is expecting more data than is available in the referenced DATA statements. Check for deleted lines, proper OPTION BASE, proper use of RESTORE, or typing errors.

38 TAB or TABXY not allowed here. The tab functions are not allowed in statements that contain a USING clause. TABXY is allowed only in a PRINT statement.

40 Improper REN, COPYLINES, or MOVELINES command. Line numbers must be whole numbers from 1 to 32 766. This may also result from a COPYLINES or MOVELINES statement whose destination line numbers lie within the source range.

41 First line number greater than second line number. Parameters out of order in a statement like SAVE, LIST, or DEL.

43 Matrix must be square. The MAT functions: IDN, INV, and DET require the array to have equal numbers of rows and columns.

44 Result cannot be an operand. Attempt to use a matrix as both result and argument in a MAT TRN or matrix multiplication.

46 Attempting a SAVE when there is no program in memory.

47 COM declarations are inconsistent or incorrect. Includes such things as mismatched dimensions, unspecified dimensions, and blank COM occurring for the first time in a subprogram.

49 Branch destination not found. A statement such as ON ERROR or ON KEY refers to a line that does not exist. Branch destinations must be in the same context as the ON...statement.

51 File not currently assigned. Attempting an ON/OFF END statement with an unassigned I/O path name.

52 Improper mass storage unit specifier. The characters used for a msus do not form a valid specifier. This could be a missing colon, too many parameters, illegal characters, etc.

53 Improper file name. File names are limited to 10 characters. Foreign characters are allowed, but punctuation is not.

54 Duplicate file name. The specified file name already exists in directory. It is illegal to have two files with the same name on one volume.

55 Directory overflow. Although there may be room on the media for the file, there is no room in the directory for another file name. Discs initialized by BASIC have room for over 100 entries in the directory, but other systems may make a directory of a different size.

56 File name is undefined. The specified file name does not exist in the directory. Check the contents of the disc with a CAT command.

58 Improper file type. Many mass storage operations are limited to certain file types. For example, LOAD is limited to PROG files and ASSIGN is limited to ASCII and BDAT files.

59 End of file or buffer found. For files: No data left when reading a file, or no space left when writing a file. For buffers: No data left for an ENTER, or no buffer space left for an OUTPUT. Also, WORD-mode TRANSFER terminated with odd number of bytes.

60 End of record found in random mode. Attempt to ENTER a field that is larger than a defined record.

62 Protect code violation. Failure to specify the protect code of a protected file, or attempting to protect a file of the wrong type.

64 Mass storage media overflow. There is not enough contiguous free space for the specified file size. The disc is full.

65 Incorrect data type. The array used in a graphics operation, such as GLOAD, is the wrong type (INTEGER or REAL).

66 INITIALIZE failed. Too many bad tracks found. The disc is defective, damaged, or dirty.

67 Illegal mass storage parameter. A mass storage statement contains a parameter that is out of range, such as a negative record number or an out of range number of records.

68 Syntax error occurred during GET. One or more lines in the file could not be stored as valid program lines. The offending lines are usually listed on the system printer. Also occurs if the first line in the file does not start with a valid line number.

72 Disc controller not found or bad controller address. The msus contains an improper device selector, or no external disc is connected.

73 Improper device type in mass storage unit specifier. The msus has the correct general form, but the characters used for a device type are not recognized.

76 Incorrect unit number in mass storage unit specifier. The msus contains a unit number that does not exist on the specified device.

77 Attempt to purge an open file. The specified file is assigned to an I/O path name which has not been closed.

78 Invalid mass storage volume label. Usually indicates that the media has not been initialized on a compatible system. Could also be a bad disc.

79 File open on target device. Attempt to copy an entire volume with a file open on the destination disc.

80 Disc changed or not in drive. Either there is no disc in the drive or the drive door was opened while a file was assigned.

81 Mass storage hardware failure. Also occurs when the disc is pinched and not turning. Try reinserting the disc.

82 Mass storage unit not present. Hardware problem or an attempt to access a left-hand drive on the Model 226.

83 Write protected. Attempting to write to a write_protected disc. This includes many operations such as PURGE, INITIALIZE, CREATE, SAVE, OUTPUT, etc.

84 Record not found. Usually indicates that the media has not been initialized.

85 Media not initialized. (Usually not produced by the internal drive.)

87 Record address error. Usually indicates a problem with the media.

88 Read data error. The media is physically or magnetically damaged, and the data cannot be read.

89 Checkread error. An error was detected when reading the data just written. The media is probably damaged.

90 Mass storage system error. Usually a problem with the hardware or the media.

93 Incorrect volume code in MSUS. The MSUS contains a volume number that does not exist on the specified device.

100 Numeric IMAGE for string item.

101 String IMAGE for numeric item.

102 Numeric field specifier is too large. Specifying more than 256 characters in a numeric field.

103 Item has no corresponding IMAGE. The image specifier has no fields that are used for item processing. Specifiers such as `# X /` are not used to process the data for the item list. Item-processing specifiers include things like `K D B A`.

105 Numeric IMAGE field too small. Not enough characters are specified to represent the number.

106 IMAGE exponent field too small. Not enough exponent characters are specified to represent the number.

107 IMAGE sign specifier missing. Not enough characters are specified to represent the number. Number would fit except for the minus sign.

117 Too many nested structures. The nesting level is too deep for such structures as FOR, SELECT, IF, LOOP, etc.

118 Too many structures in context. Refers to such structures as FOR/NEXT, IF/THEN/ELSE, SELECT/CASE, WHILE, etc.

120 Not allowed while program running. The program must be stopped before you can execute this command.

121 Line not in main program. The run line specified in a LOAD or GET is not in the main context.

122 Program is not continuable. The program is in the stopped state, not the paused state. CONT is allowed only in the paused state.

126 Quote mark in unquoted string. Quote marks must be used in pairs.

127 Statements which affect the knob mode are out of order.

128 Line too long during GET.

131 Unrecognized non-ASCII keycode. An output to the keyboard contained a CHR$(255) followed by an illegal byte.

132 Keycode buffer overflow. Trying to send too many characters to the keyboard buffer with an OUTPUT 2 statement.

133 DELSUB of non-existent or busy subprogram.

134 Improper SCRATCH statement.

135 READIO/WRITEIO to nonexistent memory location.

136 REAL underflow. The input or result is closer to zero than 10^{-308} (approximately).

140 Too many symbols in the program. Symbols are variable names, I/O path names, COM block names, subprogram names, and line identifiers.

141 Variable cannot be allocated. It is already allocated.

142 Variable not allocated. Attempt to DEALLOCATE a variable that was not allocated.

143 Reference to missing OPTIONAL parameter. The subprogram is trying to use an optional parameter that didn't have any value passed to it. Use NPAR to check the number of passed parameters.

145 May not build COM at this time. Attempt to add or change COM when a program is running. For example, a program does a LOADSUB and the COM in the new subprogram does not match existing COM.

146 Duplicate line label in context. There cannot be two lines with the same line label in one context.

150 Illegal interface select code or device selector. Value out of range.

152 Parity error.

153 Insufficient data for ENTER. A statement terminator was received before the variable list was satisfied.

154 String greater than 32 767 bytes in ENTER.

155 Improper interface register number. Value out of range or negative.

156 Illegal expression type in list. For example, trying to ENTER into a constant.

157 No ENTER terminator found. The variable list has been satisfied, but no statement terminator was received in the next 256 characters. The # specifier allows the statement to terminate when the last item is satisfied.

158 Improper image specifier or nesting images more than 8 deep. The characters used for an image specifier are improper or in an improper order.

159 Numeric data not received. When entering characters for a numeric field, an item terminator was encountered before any "numeric" characters were received.

160 Attempt to enter more than 32 767 digits into one number.

163 Interface not present. The intended interface is not present, set to a different select code, or is malfunctioning.

164 Illegal BYTE/WORD operation. Attempt to ASSIGN with the WORD attribute to a non-word device.

165 Image specifier greater than dimensioned string length.

167 Interface status error. Exact meaning depends upon the interface type. With HP-IB, this can happen when a non-controller operation by the computer is aborted by the bus.

168 Device timeout occurred and the ON TIMEOUT branch could not be taken.

170 I/O operation not allowed. The I/O statement has the proper form, but its operation is not defined for the specified device. For example, using an HP-IB statement on a non-HP-IB interface or directing a LIST to the keyboard.

171 Illegal I/O addressing sequence. The secondary addressing in a device selector is improper or primary address too large for specified device.

172 Peripheral error. PSTS line is false. If used, this means that the peripheral device is down. If PSTS is not being used, this error can be suppressed by using control register 2 of the GPIO.

173 Active or system controller required. The HP-IB is not active controller and needs to be for the specified operation.

174 Nested I/O prohibited. An I/O statement contains a user-defined function. Both the original statement and the function are trying to access the same file or device.

177 Undefined I/O path name. Attempting to use an I/O path name that is not assigned to a device or file.

178 Trailing punctuation in ENTER. The trailing comma or semicolon that is sometimes used at the end of OUTPUT statements is not allowed at the end of ENTER statements.

301 Cannot do while connected.

303 Not allowed when trace active.

304 Too many characters without terminator.

306 Interface card failure. The datacomm card has failed self-test.

308 Illegal character in data. Datacomm error.

310 Not connected. Datacomm error.

313 USART receive buffer overflow. Overrun error detected. Interface card is unable to keep up with incoming data rate. Data has been lost.

314 Receive buffer overflow. Program is not accepting data fast enough to keep up with incoming data rate. Data has been lost.

315 Missing data transmit clock. A transmit timeout has occurred because a missing data clock prevented the card from transmitting. The card has disconnected from the line.

316 CTS false too long. The interface card was unable to transmit for a predetermined period of time because Clear-To-Send was false on a half-duplex line. The card has disconnected from the line.

317 Lost carrier disconnect. Data Set Ready (DSR) or Data Carrier Detect (if full duplex) went inactive for too long.

318 No activity disconnect. The card has disconnected from the line because no data was transmitted or received for a predetermined length of time.

319 Connection not established. Data Set Ready or Data Carrier Detect (if full duplex) did not become active within a predetermined length of time.

324 Card trace buffer overflow.

325 Illegal databits/parity combination. Attempting to program 8 bits-per-character and a parity of "1" or "0".

326 Register address out of range. A control or status register access was attempted to a non-existent register.

327 Register value out of range. Attempting to place an illegal value in a control register.

328 USART Transmit underrun.

330 User-defined LEXICAL ORDER IS table size exceeds array size.

331 Repeated value in pointer. A MAT REORDER vector has repeated subscripts. This error is not always caught.

332 Non-existent dimension given. Attempt to specify a non-existent dimension in a MAT REORDER operation.

333 Improper subscript in pointer. A MAT REORDER vector specifies a non-existent subscript.

334 Pointer size is not equal to the number of records. A MAT REORDER vector has a different number of elements than the specified dimension of the array.

335 Pointer is not a vector. Only single-dimension arrays (vectors) can be used as the pointer in a MAT REORDER or a MAT SORT statement.

337 Substring key is out-of-range. The specified substring range of the sort key exceeds the dimensioned length of the elements in the array.

338 Key subscript out-of-range. Attempt to specify a subscript in a sort key outside the current bounds of the array.

340 Mode table too long. User-defined LEXICAL ORDER IS mode table contains more than 63 entries.

341 Improper mode indicator. User-defined LEXICAL ORDER IS table contains an illegal combination of mode type and mode pointer.

342 Not a single-dimension integer array. User-defined LEXICAL ORDER IS mode table must be a single-dimension array of type INTEGER.

343 Mode pointer is out of range. User-defined LEXICAL ORDER IS table has a mode pointer greater than the existing mode table size.

344 1 for 2 list empty or too long. A user-defined LEXICAL ORDER IS table contains an entry indicating an improper number of 1 for 2 secondaries.

345 CASE expression type mismatch. The SELECT statement and its CASE statements must refer to the same general type, numeric or string.

346 INDENT parameter out-of-range. The parameters must be in the range: 0 thru eight characters less than the screen width.

347 Structures improperly matched. There is not a corresponding number of structure beginnings and endings. Usually means that you forgot a statement such as END IF, NEXT, END SELECT, etc.

349 CSUB has been modified. A contiguous block of compiled subroutines has been modified since it was loaded. A single module that shows as multiple CSUB statements has been altered because program lines were inserted or deleted.

353 Data link failure.

370-399 Errors in this range are reported if a run-time Pascal error occurs in a CSUB. To determine the Pascal error number, subtract 400 from the BASIC error number. Information on the Pascal error can be found in the Pascal *User's Manual.*

401 Bad system function argument. An invalid argument was given to a time, date, base conversion, or SYSTEM$ function.

403 Copy failed; program modification incomplete. An error occurred during a COPYLINES or MOVELINES resulting in an incomplete operation. Some lines may not have been copied or moved.

427 Priority may not be lowered.

450 Volume not found—SRM error.

451 Volume labels do not match—SRM error.

453 File in use—SRM error.

454 Directory formats do not match—SRM error.

455 Possibly corrupt file—SRM error.

456 Unsupported directory operation—SRM error.

457 Passwords not supported—SRM error.

458 Unsupported directory format—SRM error.

459 Specified file is not a directory—SRM error.

460 Directory not empty—SRM error.

462 Invalid password—SRM error.

465 Invalid rename across volumes—SRM error.

471 TRANSFER not supported by the interface.

481 File locked oropen exclusively—SRM error.

482 Cannot move a directory with a RENAME operation—SRM error.

483 System down—SRM error.

484 Password not found—SRM error.

485 Invalid volume copy—SRM error.

488 DMA hardware required. HP 9885 disc drive requires a DMA card oris malfunctioning.

511 The result array in a MAT INV must be of type REAL.

600 Attribute cannot be modified. The WORD/BYTE mode cannot be changed after assigning the I/O path name.

601 Improper CONVERT lifetime. When the CONVERT attribute is included in the assignment of an I/O path name, the name of a string variable containing the conversion is also specified. The conversion string must exist as long as the I/O path name is valid.

602 Improper BUFFER lifetime. The variable designated as a buffer during an I/O path name assignment must exist as long as the I/O path name is valid.

603 Variable was not declared as a BUFFER. Attempt to assign a variable as a buffer without first declaring the variable as a BUFFER.

604 Bad source or destination for a TRANSFER statement. Transfers are not allowed to the CRT, keyboard, or tape backup on CS80 drives. Buffer to buffer or device to device transfers are not allowed.

605 BDAT file type required. Only BDAT files can be used in a TRANSFER operation.

606 Improper TRANSFER parameters. Conflicting or invalid TRANSFER parameters were specified, such as RECORDS without and EOR clause, or DELIM with an outbound TRANSFER.

607 Inconsistent attributes. Such as CONVERT or PARITY with FORMAT OFF.

609 IVAL or DVAL result too large. Attempt to convert a binary, octal, decimal, or hexadecimal string into a value outside the range of the function.

612 BUFFER pointers in use. Attempt to change one or more buffer pointers while a TRANSFER is in progress.

700 Improper plotter specifier. The characters used as a plotter specifier are not recognized. May be misspelled or contain illegal characters.

702 CRT graphics hardware missing. Hardware problem.

704 Upper bound not greater than lower bound. Applies to P2<=P1 or VIEWPORT upper bound and CLIP limits.

705 VIEWPORT or CLIP beyond hard clip limits.

708 Device not initialized.

713 Request not supported by specified device. Trying to equate color CRT characteristics with an external device; such as COLOR MAP on a plotter.

733 GESCAPE opcode not recognized. Only values 1 thru 5 can be used.

900 Undefined typing aid key.

901 Typing aid memory overflow.

902 Must delete entire context. Attempt to delete a SUB or DEF FN statement without deleting its entire context. Easiest way to delete is with DELSUB.

903 No room to renumber. While EDIT mode was renumbering during an insert, all available line numbers were used between insert location and end of program.

904 Null FIND or CHANGE string.

905 CHANGE would produce a line too long for the system. Maximum line length is two lines on the CRT.

906 SUB or DEF FN not allowed here. Attempt to insert a SUB or DEF FN statement into the middle of a context. Subprograms must be appended at the end.

909 May not replace SUB or DEF FN. Similar to deleting a SUB or DEF FN.

910 Identifier not found in this context. The keyboard-specified variable does not already exist in the program. Variables cannot be created from the keyboard; they must be created by running a program.

911 Improper I/O list.

920 Numeric constant not allowed.

921 Numeric identifier not allowed.

922 Numeric array element not allowed.

923 Numeric expression not allowed.

924 Quoted string not allowed.

925 String identifier not allowed.

926 String array element not allowed.

927 Substring not allowed.

928 String expression not allowed.

929 I/O path name not allowed.

930 Numeric array not allowed.

931 String array not allowed.

932 Excess keys specified. A sort key was specified following a key which specified the entire record.

935 Identifier is too long: 15 characters maximum.

936 Unrecognized character. Attempt to store a program line containing an improper name or illegal character.

937 Invalid OPTION BASE. Only 0 and 1 are allowed.

939 OPTIONAL appears twice. A parameter list may have only one OPTIONAL keyword. All parameters listed before it are required, all listed after it are optional.

940 Duplicate formal parameter name.

942 Invalid I/O path name. The characters after the @ are not a valid name. Names must start with a letter.

943 Invalid function name. The characters after the FN are not a valid name. Names must start with a letter.

946 Dimensions are inconsistent with previous declaration. The references to an array contain a different number of subscripts at different places in the program.

947 Invalid array bounds. Value out of range, or more than 32 767 elements specified.

948 Multiple assignment prohibited. You cannot assign the same value to multiple variables by stating X=Y=Z=0. A separate assignment must be made for each variable.

949 This symbol not allowed here. This is the general "syntax error" message. The statement you typed contains elements that don't belong together, are in the wrong order, or are misspelled.

950 Must be a positive integer.

951 Incomplete statement. This keyword must be followed by other items to make a valid statement.

961 CASE expression type mismatch. The CASE line contains items that are not the same general type, numeric or string.

962 Programmable only: cannot be executed from the keyboard.

963 Command only: cannot be stored as a program line.

977 Statement is too complex. Contains too many operators and functions. Break the expression down so that it is performed by two or more program lines.

980 Too many symbols in this context. Symbols include variable names, I/O path names, COM block names, subprogram names, and line identifiers.

982 Too many subscripts: maximum of six dimensions allowed.

983 Wrong type or number of parameters. An improper parameter list for a machine-resident function.

985 Invalid quoted string.

987 Invalid line number: must be a whole number 1 thru 32 766.

Second Byte of Non-ASCII Key Sequences

Holding the CTRL key and pressing a non-ASCII key generates a two-character sequence on the CRT. The first character is an "inverse-video" K. This table can be used to look up the key that corresponds to the second character of the sequence.

Character	Value	Key
space		[1]
!	33	STOP
"		[1]
#	35	CLR LN
$	36	ANY CHAR
%	37	CLR→END
&	38	Select
'	39	Prev
(	40	SHIFT-TAB
)	41	TAB
*	42	INS LN
+	43	INS CHR
,	44	Next
-	45	DEL CHR
.	46	Ignored
/	47	DEL LN
0	48	k0
1	49	k1
2	50	k2
3	51	k3
4	52	k4
5	53	k5
6	54	k6
7	55	k7
8	56	k8
9	57	k9
:	58	SHIFT-system f6 [2]
;	59	SHIFT-system f7 [2]
<	60	←
=	61	RESULT
>	62	→
?	63	RECALL
@	64	SHIFT-RECALL
A	65	PRT ALL
B	66	BACK SPACE
C	67	CONTINUE
D	68	EDIT
E	69	ENTER
F	70	DISPLAY FCTNS
G	71	SHIFT-→
H	72	SHIFT-←
I	73	CLR I/O
J	74	Katakana Mode
K	75	CLR SCR
L	76	GRAPHICS
M	77	ALPHA
N	78	DUMP GRAPHICS
O	79	DUMP ALPHA

Character	Value	Key
P	80	PAUSE
Q		[1]
R	82	RUN
S	83	STEP
T	84	SHIFT-↓
U	85	CAPS LOCK
V	86	↓
W	87	SHIFT-↑
X	88	EXECUTE
Y	89	Roman Mode
Z		[1]
[	91	CLR TAB
\	92	▶
]	93	SET TAB
^	94	↑
_	95	SHIFT-▶
`		[1]
a	97	k10
b	98	k11
c	99	k12
d	100	k13
e	101	k14
f	102	k15
g	103	k16
h	104	k17
i	105	k18
j	106	k19
k	107	k20
l	108	k21
m	109	k22
n	110	k23
o	111	SHIFT-system f1 [2]
p	112	SHIFT-system f2 [2]
q	113	SHIFT-system f3 [2]
r	114	SHIFT-system f4 [2]
s	115	SHIFT-user f1 [2]
t	116	SHIFT-user f2 [2]
u	117	SHIFT-user f3 [2]
v	118	SHIFT-user f4 [2]
w	119	SHIFT-user f5 [2]
x	120	SHIFT-user f6 [2]
y	121	SHIFT-user f7 [2]
z	122	SHIFT-user f8 [2]
}	123	System
|	124	Menu
{	125	User
~	126	SHIFT-Menu
▒		[1]

1 These characters cannot be generated by pressing the CTRL key and a non-ASCII key. If one of these characters follows CHR$(255) in an output to the keyboard, an error is reported (`Error 131 Bad non-alphanumeric keycode.`).

2 System and user refer to the softkey menu which is currently active.

US ASCII Character Codes

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
NUL	0	00000000	000	00	
SOH	1	00000001	001	01	GTL
STX	2	00000010	002	02	
ETX	3	00000011	003	03	
EOT	4	00000100	004	04	SDC
ENQ	5	00000101	005	05	PPC
ACK	6	00000110	006	06	
BEL	7	00000111	007	07	
BS	8	00001000	010	08	GET
HT	9	00001001	011	09	TCT
LF	10	00001010	012	0A	
VT	11	00001011	013	0B	
FF	12	00001100	014	0C	
CR	13	00001101	015	0D	
SO	14	00001110	016	0E	
SI	15	00001111	017	0F	
DLE	16	00010000	020	10	
DC1	17	00010001	021	11	LLO
DC2	18	00010010	022	12	
DC3	19	00010011	023	13	
DC4	20	00010100	024	14	DCL
NAK	21	00010101	025	15	PPU
SYNC	22	00010110	026	16	
ETB	23	00010111	027	17	
CAN	24	00011000	030	18	SPE
EM	25	00011001	031	19	SPD
SUB	26	00011010	032	1A	
ESC	27	00011011	033	1B	
FS	28	00011100	034	1C	
GS	29	00011101	035	1D	
RS	30	00011110	036	1E	
US	31	00011111	037	1F	

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
space	32	00100000	040	20	LA0
!	33	00100001	041	21	LA1
"	34	00100010	042	22	LA2
#	35	00100011	043	23	LA3
$	36	00100100	044	24	LA4
%	37	00100101	045	25	LA5
&	38	00100110	046	26	LA6
'	39	00100111	047	27	LA7
(	40	00101000	050	28	LA8
)	41	00101001	051	29	LA9
*	42	00101010	052	2A	LA10
+	43	00101011	053	2B	LA11
,	44	00101100	054	2C	LA12
–	45	00101101	055	2D	LA13
.	46	00101110	056	2E	LA14
/	47	00101111	057	2F	LA15
0	48	00110000	060	30	LA16
1	49	00110001	061	31	LA17
2	50	00110010	062	32	LA18
3	51	00110011	063	33	LA19
4	52	00110100	064	34	LA20
5	53	00110101	065	35	LA21
6	54	00110110	066	36	LA22
7	55	00110111	067	37	LA23
8	56	00111000	070	38	LA24
9	57	00111001	071	39	LA25
:	58	00111010	072	3A	LA26
;	59	00111011	073	3B	LA27
<	60	00111100	074	3C	LA28
=	61	00111101	075	3D	LA29
>	62	00111110	076	3E	LA30
?	63	00111111	077	3F	UNL

STD-LL-60182

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
@	64	01000000	100	40	TA0
A	65	01000001	101	41	TA1
B	66	01000010	102	42	TA2
C	67	01000011	103	43	TA3
D	68	01000100	104	44	TA4
E	69	01000101	105	45	TA5
F	70	01000110	106	46	TA6
G	71	01000111	107	47	TA7
H	72	01001000	110	48	TA8
I	73	01001001	111	49	TA9
J	74	01001010	112	4A	TA10
K	75	01001011	113	4B	TA11
L	76	01001100	114	4C	TA12
M	77	01001101	115	4D	TA13
N	78	01001110	116	4E	TA14
O	79	01001111	117	4F	TA15
P	80	01010000	120	50	TA16
Q	81	01010001	121	51	TA17
R	82	01010010	122	52	TA18
S	83	01010011	123	53	TA19
T	84	01010100	124	54	TA20
U	85	01010101	125	55	TA21
V	86	01010110	126	56	TA22
W	87	01010111	127	57	TA23
X	88	01011000	130	58	TA24
Y	89	01011001	131	59	TA25
Z	90	01011010	132	5A	TA26
[	91	01011011	133	5B	TA27
\	92	01011100	134	5C	TA28
]	93	01011101	135	5D	TA29
^	94	01011110	136	5E	TA30
_	95	01011111	137	5F	UNT

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
`	96	01100000	140	60	SC0
a	97	01100001	141	61	SC1
b	98	01100010	142	62	SC2
c	99	01100011	143	63	SC3
d	100	01100100	144	64	SC4
e	101	01100101	145	65	SC5
f	102	01100110	146	66	SC6
g	103	01100111	147	67	SC7
h	104	01101000	150	68	SC8
i	105	01101001	151	69	SC9
j	106	01101010	152	6A	SC10
k	107	01101011	153	6B	SC11
l	108	01101100	154	6C	SC12
m	109	01101101	155	6D	SC13
n	110	01101110	156	6E	SC14
o	111	01101111	157	6F	SC15
p	112	01110000	160	70	SC16
q	113	01110001	161	71	SC17
r	114	01110010	162	72	SC18
s	115	01110011	163	73	SC19
t	116	01110100	164	74	SC20
u	117	01110101	165	75	SC21
v	118	01110110	166	76	SC22
w	119	01110111	167	77	SC23
x	120	01111000	170	78	SC24
y	121	01111001	171	79	SC25
z	122	01111010	172	7A	SC26
{	123	01111011	173	7B	SC27
|	124	01111100	174	7C	SC28
}	125	01111101	175	7D	SC29
~	126	01111110	176	7E	SC30
DEL	127	01111111	177	7F	SC31

Subject Index

a

ABS 82
Accessing Directories 246
Accessing Files 222
Accessing Mass Storage 210
Accuracy 76
Accuracy of the Clock 275
ACS 82
ALLOCATE 74, 120
ALPHA Key 331
ALPHA OFF 331
ALPHA ON 331
Angle Functions 83
ANY CHAR Key 40, 144
AP2.0 379
Appending Program Lines 30
Arrays 74
  Copying 92
  Declaring 74
  Dimensioning 85
  Indexing 321
  Operations 85
  Operators 96
  Reordering 99
  Sorting 100
  String 120
ASCII Character Codes 394
ASCII Character Set 140
ASCII Files 206
ASN 82
ASSIGN @ 223
ATN 82
Auto Line Numbering 7
AUTOST 35
Autostart 35
Autostart on SRM 36
AXES 336

b

BASE 82, 91
Base Conversion 138
BASIC 2.0 379
BDAT Files 203
  Reading 225
  Writing 225
Benchmarking 319
BINAND 82
Binary Tree 186
BINCMP 82
BINEOR 82
BINIOR 82
BINs 37
  Deleting from Memory 43
  Loading 37
  Scratching 37, 43
BIT 82
Blank Lines 283
Boolean Arrays 98
Boundary Conditions 300
Boxing the Screen 333
BUBBLE 212
Bubble Memory 220
Bugs 299, 307

c

CALL 169
Calling a Subprogram 169
CASE 57
Case Conversion 130
CAT 27, 246
Catalog Header, Suppressing 250
Cataloging the Disc 247
Cataloging, Skipping Files 251
CHANGE 22
Character Height 337
Character Set, Extended 143
Character Set, Highlights 143
Character Width 337

CHR$ 127
Clearing the Computer 43
Clearing the CRT 282
CLIP 339
Clipping 338
Clock 265
  Accuracy 275
  Events 276
  Setting 268, 270
Closing an I/O Path 223
CLR I/O 314
CLR I/O Key 17
Color Graphics 347
Color Lines 349
COM 32, 120
COM Blocks 173
Command 6
Comments 11, 13, 317
Common 316
Common and GET 34
Comparing REAL Numbers 301
Comparision Operators 79
Concatenation, Strings 121
Conditional Branching 53
Conditional Execution 51
Configurating a System 37
Constants 325
Context Switching 176
CONTINUE 47
CONTROL 283
Control Characters 257
Control Characters, Displaying 140
COPY 38, 244
Copying
  Arrays 92
  Files 38, 244
  Program Segments 21
  Volumes 244
COPYLINES 21
COS 82
CREATE ASCII 246
CREATE BDAT 225
Cross References 26
CRT 82, 256
  Clearing 282
  Size 331, 341
CS80 212
CSIZE 337
CSUBs 379, 380
CSUM 116

d

DATA 92, 194
Data Files 318
  Structure 203
Data
  Input 194
  Pointer, Moving 197
  Retrieval 193
  Storage 193, 315, 316
  Structure 185
  Type Conversion 74
  Types, Numeric 73
DATE 82, 266
DATE$ 265
Deactivating Events 69
Debugging 307
Declaring Arrays 74
Declaring Variables 74
DEF 168
Default Dimensioning 119
Defined Records 225
Defining Softkeys 39
Degrees 83
DEL Command 10
Deleting Lines 10
Deleting Subprograms 23, 181, 182
DELSUB 23, 181
DET 82, 111
Determinant of a Matrix 111
Device Selectors 212, 253, 256
Device Type 211
DIGITIZE 355
Digitizing 355
DIM 120, 74
Dimension Table 315
Dimensioning an Array 85
Directories, Accessing 246
Directories, Reading 246
Disabling Events 69, 71
Disc
  Cataloging 247
  Copying 244
  Directory 201
  Initialization 208
  Interleave 200
  Labels 209
  Structure 198
Disc Drives, External 214

Disc Drives, Internal 214
Displaying Control Characters 140
Displays 282
DOT 82, 106
Double Subscripted Substrings 123
DRAW 332
Drawing 332
  Axes 336
  Grids 336
  in Color 349
DROUND 81, 82, 84
DUMP DEVICE IS 344
DVAL 82, 138
DVAL$ 138
Dyadic Operators 79

e

Edit 7
EDIT KEY 39
Edit Mode, Exiting 14
Editing Softkeys 39
Editing Subprograms 182
Editor 7
Enabling Events 66
END 46
END IF 54
END LOOP 64
END WHILE 62
End-of-File 240
End-Of-File Pointers 228
End-of-Record 240
Ending Functions 183
Ending Subprograms 183
ENTER 6, 238
Entering a Single Item 293
Entering Program Lines 8
EOF Pointers 228
EPROM 212
ERRL 302
ERRM$ 302
ERRN 302
Error Messages 383
Error Numbers 302
Error Trapping 302
Errors 299, 383
Errors, Operator 300
Escape Code Sequences 258
Event-Initiated Branching 45, 66
Events 66
  Deactivating 69
  Disabling 69
  Enabling 66
EXEC 6
EXECUTE 6
Executing a Subprogram 169
EXIT IF 64
Exiting Edit Mode 14
EXP 82
Expressions, Evaluating 77
Extended Character Set 143
External Disc Drives 214
External Printers 257

f

Files
  Accessing 222
  ASCII 206
  BDAT 203
  Copying 38, 244
  Data 318
  Names 28, 202
  Opening 222
  Plotting to 346
  Program 318
  Protecting 38, 242
  Purging 38, 245
  Renaming 38
  Types 202
FIND 21
FN 168
FNEND 183
FOR NEXT 59
Formatted Printing 259
FRACT 82
FRAME 333
Function or Subprogram 167
Functions
  Ending 183
  String 125, 129
  User-Defined 165

g

GDUs 333
GET 30
GINIT 332, 342
GLOAD 341
GOSUB 48
GOTO 48
Graphic Display Units 333, 336
Graphic Units 336, 340
Graphics 331
  Color 347
  Initializing 332, 342
  Interactive 354
  Monitors 331
  Output Devices 342
  Saving an Image 341
  Storing 351
GRAPHICS Key 331
GRAPHICS OFF 283, 331
GRAPHICS ON 331
GRID 336
GSTORE 341

h

Halting Program Execution 46
Hard Clip Limits 338
Hardware 382
Hierarchy 121
Hierarchy of Numeric Operating 77
Highlight Characters 143
HPGL 345

i

I/O Path 222
  Closing 223
  Opening 222
Identifiers 317
Identity Matrix 107
IDN 107
IDRAW 334
IF THEN 51
IF THEN ELSE 55
Ill-Conditioned Matrices 112
IMAGE 260
Image Specifiers, Numeric 261
Image Specifiers, String 262
Images 260
IMOVE 334
Implicit Dimensioning 90
Incremental Moves 334
INDENT 23
Indenting a Program 23
Initializing a Disc 208
Initializing Graphics 332, 342
Inputting Multiple Fields 296
INSERT LINE Key 9
Inserting Lines 9
Inserting Subprograms 182
INT 82
INTEGER 73
Integer Numbers 204
INTEGER Variables 321
Interactive Graphics 353
Interleave on Discs 200
INTERNAL 212
Internal Disc Drives 214
Internal Numeric Formats 75
Interval Timing 274
Introduction 1
INV 108
Inverse Matrix 107
IPLOT 334
Isotropic View 340
IVAL 82

k

KBD 39, 82, 256
Keyboard Input 292
Keyboards 8
Keyword 5
Knob, Using 68
KNOBX 69
KNOBY 69, 380

l

LABEL 337
Label Location 337
Labels, Disc 209
LDIR 337
LEN 125
Length of a String 119, 125
Lexical Order 139
LEXICAL ORDER IS 130, 139
Lexical Order, Predefined 145
Lexical Order, User-Defined 156
Lexical Tables 146
LEX_AID 158
LGT 82
Line Rotation 335
LINE TYPE 337
Linear Program Flow 45, 46
LINPUT 296
LIST 11
LIST KEY 41
Listing a Program 11
Live Keyboard 16, 308
LOAD 28, 34
LOAD KEY 41
Loading BINs 37
Loading Softkeys 41
Loading Subprograms 23, 180
LOADSUB 23, 180
LOG 83
Logical Comparisions 326
LOOP 59, 63
Loop Counter 60
Loops 322
LORG 337
LWC$ 130

m

Main Program 5
Mapping 356
Mass Memory Performance 318
Mass Storage 198
Mass Storage Access 210
MASS STORAGE IS 214
Mass Storage Unit Specifier 27
Mass Storage, Non-Disc 220
MAT 93
MAT Functions 132
MAT REORDER 99
MAT SORT 100
Math Hierarchy 77
Mathematical Operations 321
Matrix 103
  Determinant 111
  Identity 107
  Ill-Conditioned 112
  Inverse 107
  Multiplication 103
  Singular 110
  Summing Columns 116
  Summing Rows 116
  Transposition 115
MAX 83
MAXREAL 83
Media Specifiers 211
MEMORY 212
Memory, Saving 329
Menues 285
Merging Subprograms 182
MIN 83
MINREAL 83
Monadic Operators 79
MOVE 332
MOVELINES 20
Moving a Data Pointer 197
Moving EOF Pointers 228
Moving Program Segments 20
Moving the Pen 332
MSUS 27, 211
Multiple Fields Input 296

n

Naming Files 28
Naming Subprograms 165
Nesting Structures 54
Non-ASCII Keys *284*, 393
Non-ASCII Keystrokes 40
Non-Disc Mass Storage 220
NPAR 172
Number Base Conversion 138
Numbers, Comparing 301
Numeric
  Accuracy 76
  Computation 73
  Data Types 73
  Formats, Internal 75
  Functions *82*
  Image Specifiers 261
  Precision 76
Numeric to String Conversion 127

o

OFF-event 70
ON CYCLE 276, 66
ON DELAY 276, 66
ON END 240, 66
ON EOR 66
ON EOT 66
ON ERROR 302, 303, 66
ON INTR 66
ON KBD 66
ON KEY 66, 67
ON KNOB 66, 68
ON SIGNAL 66
ON Statement 57
ON TIME 276, 66
ON TIMEOUT 66
ON-event 66
Opening a File 222
Opening an I/O Path 222
Operator Errors 300
Operator Hierarchy 77
Operators 79
  Comparision 79
  Dyadic 79
  Monadic 79
OPTION BASE 86
Optional Parameters 171
OUTPUT 229, 230
OUTPUT KBD 283
Overhead 315

p

Parameters 170
Parameters, Optional 171
PAUSE 47
PAUSE Key 17
Pausing a Program 17
PDEV 20
PEN 332
Pen Control 334
Pen, Moving 332
Pen, Moving Incremental 334
Performance 318
PHYREC 379, 381
PI 83
PIVOT 335
PLOT 332
PLOTTER IS 342
PLOTTER IS file 346
Plotting to a File 346
Polynomial Evaluations 325
POS 125
Position of a Substring 125
Powerfail 275
Precision 76
Prerun 15
Primary Address 254
PRINT 259
PRINT USING 260
PRINTALL IS 256, 313
PRINTER IS 254
Printer Switch Setting 254
Printers 253
  Control Characters 257
  Escape Codes 258
  External 257
Printing, Formatted 259
PROG Files 28
Program Counter 45
Program Execution 15, 310
Program Execution, Selection 51
Program Files 318
Program Flow
  Linear 45
  Repetition 45
  Selection 45
  Sequence 45
Program Line 5
Programming a LOAD 35
Programming GET 31
Programs
  Recording 28
  Replacing 29
  Retrieving 27, 30
  Storing 27
Prompts 282
PROTECT 242, 38
Protecting Files 38, 242
PROUND 83, 84
PRT 256, 83
PURGE 245, 38
Purging Files 38, 245

r

Radians. 83
RAM Volumes 220
Random ENTER 238
Random Numbers 84
Random OUTPUT 234
RANK 83
RATIO 336
RE-SAVE 29
RE-STORE 29
RE-STORE BIN 381
RE-STORE KEY 41
READ 92, 194
READ LOCATOR 358
Reading BDAT Files 225
Reading Data From BDAT Files 237
Reading Directories 246
REAL 73
REAL Number Comparisions 301
Real Numbers 204, 324
Real-Time Clock 265
Recalling Lines 10
Record Lengths 226
Recording a Program 28
RECOVER 176
Recursion 184
REDIM 95
Redimensioning Arrays, Automatic 93
Redimensioning Arrays, Explicit 95
Relational Operations 121
Relative Moves 335
REM 12
REMOTE 212
REN Command 10
RENAME 38
Renaming a File 38
Renumbering a Program 10
Reordering Arrays 99, 135
REPEAT UNTIL 59, 61
Repeating a String 129
Repetition 59
Replacing Programs 29
RES 83
RESET Key 17
Resources 315
RESTORE 197
Retrieving Programs 27, 30
RETURN 48
Returning from a Subprogram 49
REV$ 129
Reversing a String 129
RND 83, 84
ROTATE 83
Rotating Lines 335
Rounding 84
Rounding Numbers 81
RPLOT 335
RPT$ 129
RSUM 116
RUN 15
Run Light 17
Run-time 16
Running a Program 15

s

SAVE 28
Saving an Image 341
Saving Memory 329
Saving Time 327
SC 83
Scalar Expressions 77
SCRATCH 43
SCRATCH A 43
SCRATCH BIN 43
SCRATCH C 43
SCRATCH KEY 43
Scratching BINs 37
Screen Width 282
Search and Replace 21
Searching for Strings 136
SECURE 243
Securing Program Lines 243
SELECT 56
SELECT CASE 57
Serial ENTER 237
Serial OUTPUT 229
SET ECHO 358
SET TIME 268
SET TIMEDATE 266
Setting the Clock 268, 270
SGN 83
SHIFT 83
SHOW 340
Simple Branching 48
SIN 83
Single Byte Access 239
Single-Subscripted Substrings 122
Singular Matrices 110
SIZE 83, 91
Soft Clip Limits 338

Softkeys 39, 285
  Defining 42
  Definitions 39, 41
  Deleting from Memory 43
  Editing 39
  Files 41
  Listing 41
  Loading 41
Solving Simultaneous Equations 108
Sorting
  Arrays 100
  by a Vector 134
  by Substrings 133
  Strings 131
SQR 83
SRM, Autostart 36
Statement 5
Statements, New 380
STEP Key 310
Stepping 310
STOP 46
STOP Key 17
Stopping a Program 17
STORE 28
STORE KEY 41
STORE SYSTEM 37, 379
Storing
  Systems 37
  Data 194
  Data in Variables 194
  Graphics 351
  Programs 27
  Strings 120
String to Numeric Conversion 126
Strings 80, 119
  Arrays 120
  Concatenation 121
  Default Dimensioning 119
  Evaluation Hierarchy 121
  Functions 125, 129
  Image Specifiers 262
  Length 119, 125
  Relational Operations 121
  Repeat 129
  Reverse 129
  Sorting 131
  Storing 120
  Trimming 129
SUBEND 183
Subprogram or Function 167
Subprograms 5
  Calling 169
  Deleting 181
  Editing 182
  Ending 183
  Executing 169
  Inserting 182
  Libraries 23, 180
  Loading 180
  Merging 182
  Naming 165
  RECOVER 177
  Returning from 49
  Softkeys 177
  Speed 178
  User-Defined 165
  Variables 177
Substrings 122
  Double Subscripts 123
  Position 125
  Single Subscripts 122
  Sorting 133
SUM 83, 94
Summing Columns in Arrays 116
Summing Rows in Arrays 116
Suppressing a Catalog Header 250
Switching Context 176
Symbol Table 315
Syntax 9
Syntax Checking 9
System Configuration 37
SYSTEM$("LEXICAL ORDER IS") 145
SYSTEM$("KEYBOARD LANGUAGE") 145
Systems, Storing 37

t

TAB 259
TABXY 259
TAN 83
Time 265
TIME 83, 266
TIME$ 265
Time, Saving 329
TIMEDATE 265
Timing Interval 274
Token Table 315
TRACE ALL 311
TRACE OFF 313
TRACE PAUSE 313
Tracing 311
TRACK IN ON 357

Tracking 357
Transporting Programs 379
Transposing Matrices 115
Trapping Errors 302
TRIM$ 129
Trimming a String 129
Type Conversion 324
Typing Aids 39

u

UDUs 340
Unit Number 213
UNTIL 61
UPC$ 130
Upgrading BASIC Programs 379
Upper and Lower Case 130
Uppercase and Lowercase 9
User Defined Units 340
User-Defined
  Functions 165
  Lexical Order 156
  Subprograms 165

v

VAL 126
VAL$ 127
Variables 317
Variables, Declaring 74
VIEWPORT 341
Volume Label 201
Volume Number 213
Volumes, Copying 244

w

WAIT 47
WHILE 59, 62
WINDOW 340
Writing Data 229
Writing to BDAT Files 225

x

XREF 20, 26

Manual Comment Sheet Instruction

If you have any comments or questions regarding this manual, write them on the enclosed comment sheets and place them in the mail. Include page numbers with your comments wherever possible.

If there is a revision number, (found on the Printing History page), include it on the comment sheet. Also include a return address so that we can respond as soon as possible.

The sheets are designed to be folded into thirds along the dotted lines and taped closed. Do not use staples.

Thank you for your time and interest.

MANUAL COMMENT SHEET

BASIC 3.0 Programming Techniques
for the HP 9000 Series 200

98613-90010

May 1984

Update No. _____

(See the Printing History in the front of the manual)

Name: _____

Company: _____

Address: _____

_____

Phone No: _____

fold _____ fold

fold _____ fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Fort Collins Systems Division
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525



Part No. 98613-90010
E 0584
Microfiche No. 98613-99010

Printed in U.S.A.
First Edition, May 1984